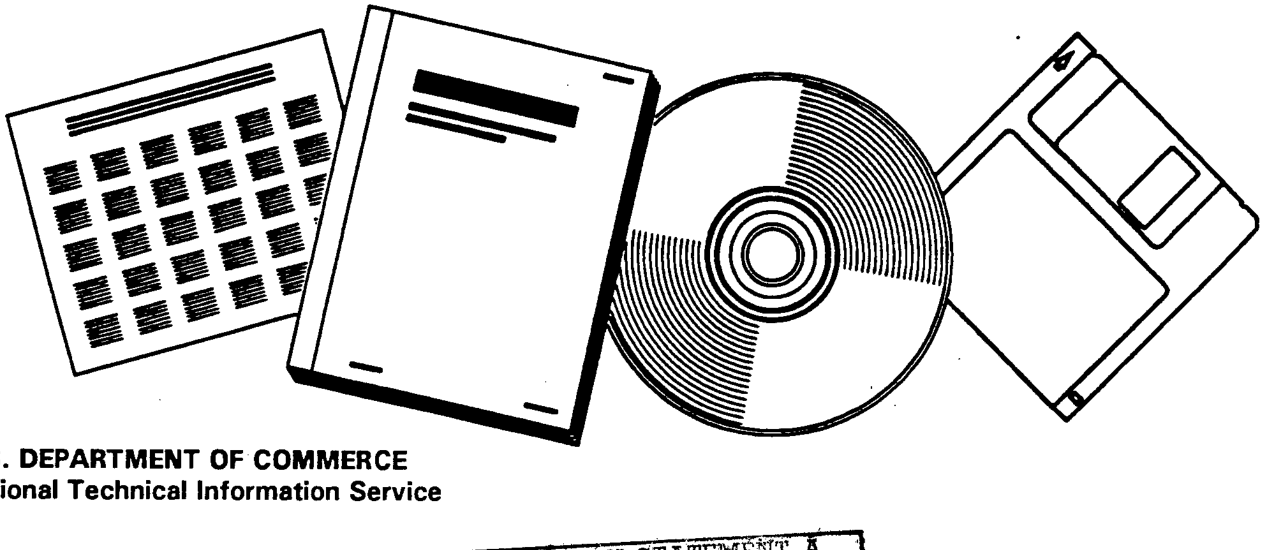
PHYSICAL DATABASE DESIGN METHODOLOGY USING THE PROPERTY OF SEPARABILITY

STANFORD UNIV., CA

MAY 83



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

May 1983

Report No. STAN-CS-83-968

PB96-149778

A Physical Database Design Methodology Using the Property of Separability

by

Kyu-Young Whang

Department of Computer Science

Stanford University
Stanford, CA 94305

REPRODUCED BY: NTIS
U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22161

A Physical Database Design Methodology Using the Property of Separability

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Kyu-Young Whang

May 1983

I certify that I have read this thesis and that in my
opinion it is fully adequate, in scope and quality, as
a dissertation for the degree of Doctor of Philosophy

Gio Wiederhold

(Principal Advisor, Computer Science)

I certify that I have read this thesis and that in my
opinion it is fully adequate, in scope and quality, as
a dissertation for the degree of Doctor of Philosophy

John T. Gill III

I certify that I have read this thesis and that in my
opinion it is fully adequate, in scope and quality, as
a dissertation for the degree of Doctor of Philosophy

Daniel Sagalowicz

(SRI International)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

A new approach to the multifile physical database design is presented. Most previous approaches towards multifile physical database design concentrated on developing cost evaluators for given designs. To accomplish the optimal physical design, however, these approaches had to rely on the designer's intuition or on exhaustive search, which is practically infeasible even for moderate-sized databases.

In our approach we develop a theory called *separability* to partition the entire database design problem into collective subproblems. Straightforward heuristics are employed to incorporate the features that cannot be included in the formal theory. This approach is somewhat formal, deliberately avoiding excessive reliance on heuristics. Our purpose is to render the whole design phase manageable and to facilitate understanding of the underlying mechanisms.

We develop a design methodology for relational database systems based on the theory. First, we set up a *basic design phase* in accordance with a formal method that includes a large subset of practically important join methods and then, using heuristics, extend the design procedure to include other join methods as well.

We show that the theory of separability can be applied to network model databases as well. In particular, we show that a large subset of practically important access structures that are available in network model database systems satisfies the conditions for separability.

As an application to the above theory, we propose three physical database design algorithms for relational database systems. These algorithms have been fully implemented in the Physical Database Design Optimizer (PhyDDO) in about 6000 lines of Pascal code and tested for their validation. The results show that the solutions generated by the design algorithms do not significantly deviate from the optimal solutions. For the implementation of these design algorithms an extensive set of cost formulas for queries, update, deletion, and insertion transactions have been developed.

Index selection is an important subproblem of physical database design. Index selection algorithms for relational databases are introduced and tested for their validation. The results show that these heuristic algorithms do not produce significant deviations from the optimal solutions.

Finally, we introduce a closed noniterative formula for estimating the number of block accesses. This formula, an approximation of Yao's exact formula, provides significant improvements in both speed of evaluation and accuracy compared with earlier formulas developed by Yao and Cardenas.

In summary, important issues on multifile physical database design are investigated in this dissertation. The proposed methodology is consolidated through extensive implementation and validation procedures. We believe that our approach can enable substantial progress to be made in the optimal design of multifile physical databases.

Foreword

This dissertation consists of three components: main chapters, major appendices, and minor appendices. The major appendices consists of six papers[1] that either have been published, accepted, or submitted for publication. The main chapters are a continuous summary of the research presented in the major appendices. Some topics that are not fully discussed in the papers are also included in the main chapters. Appendix A has been published in the Proceedings of the Seventh International Conference on Very Large Databases, Cannes, France, September 1981, and also has been submitted for publication to an IEEE journal. Appendix B has been published in the Proceedings of the Eighth International Conference on Very Large Databases, Mexico City, Mexico, September 1982. Appendix C has been accepted by the Communications of the ACM. Appendices D, E, and F have been submitted to publications such as IEEE or ACM Transactions. Minor appendices (Appendix G to Appendix J) supplement the topics discussed only partially in the main chapters and appendices. The work described in the first three papers, coauthored by Professor Gio Wiederhold and Dr. Daniel Sagalowicz, has been performed by the author as part of his dissertation research under the careful supervision of the two coauthors.

This work was supported by the Defense Advanced Research Project Agency, under the KBMS Project, Contract No. N39-80-G-0132 and N39-82-C-0250.

[1] In this report the first three papers are omitted from the original dissertation since they have already been published elsewhere.

To my parents, wife and son

Acknowledgments

I am greatly indebted to Professor Gio Wiederhold, my principal advisor, for his careful guidance and support through my years at Stanford. He introduced me to this field of research and provided much thoughtful and critical advice on the progress. His full confidence in my work and strong commitment as a supervisor was the major driving force during the hard years of my dissertation research. Despite his incredibly busy schedule, he never spared any effort in his personal and professional support. I am honored to be with him.

I owe a great deal to Dr. Daniel Sagalowicz who supervised my research from the beginning with constant faith and enthusiasm. The numerous discussions I had with him could not be more pleasant and more illuminating. His firm confidence in my work and clocklike accuracy in his advice served as a compass during the sail through the rough sea of uncertainty.

I wish to thank Professor John T. Gill III, my associate and program advisor, for his guidance over the entire period of my study at Stanford. Especially his advice at the earlier stages of my education at Stanford was indispensable in establishing the right direction to pursue.

Many research colleagues contributed to forming and refining the ideas of this work. Discussions with Ramez El-masri and Sheldon Finkelstein were particularly helpful at an earlier stage of the research. Dr. Paolo Tiberio and Dr. Jerry Kaplan carefully read several of my papers and made constructive comments. Dr. Mario Schkolnick and Dr. Won Kim read some of my papers and contributed useful comments. I also received invaluable help from other members of the KBMS Project: they are Neil Rowe, Jonathan King, Mohammad Olumi, Tom Rogers, Jim Davidson, and Arthur Keller. I am grateful to Fred Chow who helped me learn the techniques of professional programming while working together in the S-1 Project. I appreciated his brotherly friendship. I always enjoyed chats with Edwin Pednault, my long time office mate. I appreciated the help from Jayne Pickering in revising the English of all my papers.

I also want to thank Mrs. Voy Wiederhold, Professor & Mrs. McWhorter, Mr. & Mrs. Willards, and the Watneys for their help, encouragement and friendship.

This dissertation is dedicated to my parents, Sam-Hyun and Young-Hae; my wife, Jung-Hae and my son, Eui-Jong. To me, throughout my life, their love is fundamental.

Table of Contents

List of Figures

1. Introduction

1.1 Issues on Physical Database Design

The problem of physical database design is concerned with finding designing the underlying storage structures that support the logical databases. Since a good design of the physical database has a vital influence on the database performance, the physical design problem has long been an object of intense research and interest. Typically, the research in this area has been performed in several directions: file modelling and selection, access structure selection, and index selection. Each area will be briefly surveyed in the following subsections.

Before proceeding, we define two new terms that play important roles throughout the development of the thesis. First, we define the term *access structures* as the features that a particular database management system (DBMS) provides for the physical database design. For instance, access structures can be indexes, hashed organization, clustering of the records, etc. Second, we define the term *access configuration* of a logical object—such as a relation in relational database systems, a record type in network model database systems, or an entire database—to mean the aggregate of access structures specified to support that logical object. Thus, the access configuration is an abstraction of the physical database.

A related problem that has a significant effect on database performance is query optimization. Query optimization seeks the optimal sequence of access operations for processing a specific query given a certain access configuration of the underlying physical database. Since query optimization has a close relationship with physical database design, we first introduce a short survey on this subject.

1.1.1 Query optimization

The query optimization problem has been most often addressed in the context of relational database systems. The *optimizer* is a component of a DBMS that automatically translates the transactions expressed in high-level query languages—such as relational calculus or relational algebra languages—to an optimal (or suboptimal) sequences of access operations to process the transactions. In a DBMS having an optimizer the user need not know the physical structure of the database. Instead, the optimizer estimates the cost of each possible alternative for processing the transaction based on the given physical structure of the database and figures out the minimum-cost sequence of access operations. Various algorithms for query optimization have been extensively studied. Smith and Chang [SMI 75], and Pecherer [PEC 75] studied optimization of transactions expressed in relational algebra. Various join methods were investigated in [GOT 75], [BLA 76], and [YAO 79]. Detailed optimization algorithms for some existing database management systems were also introduced. One for System R [AST 76], based on the modified branch-and-bound technique, was presented in [SEL 79]. An algorithm for INGRES [STO 76] based on decomposing a multivariable query into a sequence of one-variable queries was presented in [WON 76]. An improved version of the INGRES optimization strategy appeared in [KOO 82]. Query optimization has also been investigated in systems where databases are distributed over multiple processors. Hevner and Yao [HEV 79] developed an optimization algorithm for distributed databases using the optimization criterion of minimizing the data communication cost between different sites. An optimization strategy for SDD-1 (System for Distributed Databases) using semijoins was presented in [GOO 79].

1.1.2 File modelling and selection

This problem addresses selecting appropriate file structures for a given collection of records and user requirements. There are several levels of approach towards this problem. The first level deals with specific file structures such as ISAM files and their implementations in detail [SEN 69]. The second deals with specific file structures such as inverted files or multilists, but ignore implementation details such as hardware considerations [CAR 75]. The third attempts to model most of the existing file structures with a unifying model and provides a generalized cost function of accesses.

The pioneering work in developing unifying models was done by Hsiao and Harary [HSI 70] who formalized the structure of the file as two level structures consisting of a directory and a set of records. Severance [SEV 75] refined the model by introducing two types of pointers between elements of the structure: successor pointers and data pointers. Yao [YAO-a 77] subsequently generalized both models by allowing multilevel directory structure. A unifying model for multifile databases was developed by Batory [BAT 82]. This model exploits the notion of database decomposition in which a database is modelled by a set of simple files and a set of link sets interconnecting these simple files.

In a different approach, instead of using a unifying model of different file structures, Severance and Carlis [SEV 77] developed a simple taxonomy of various file structures. Using this taxonomy, appropriate file structures can readily be chosen from the characteristics of the application which is expressed in terms of average quantity of records retrieved, required speed of response, and volume of on-line updates.

In most research in file modelling, the emphasis was on developing cost functions that evaluate the cost of processing transactions acting upon a database having a certain structure. In these approaches, however, selection of the optimal file structure can only be done according to the designer's intuition or by trial-and-error. Automatic selection of the optimal file structure for large multifile databases will be addressed in the next subsection.

1.1.3 Access structure selection

The access structure selection problem addresses finding an access configuration that gives the best performance. A major premise in this problem is the existence of a database management system which provides access structures to be utilized in physical database design. In particular, we are not concerned with designing access structures themselves; it is assumed that they have already been implemented according to the specific technique employed by the DBMS considered.

A straightforward approach to this problem is to design a cost evaluator that produces the total cost of processing transactions acting upon a specific access configuration. Using this cost evaluator, an optimal access configuration can be found by exhaustively searching through all possible access configurations–by designer's intuition or by trial-and-error. Teorey and Oberlander [TEO 78] presented a database design evaluator as a design aid to Honeywell's IDS [HON 71]. Gerritsen and Gambino [GER 77], [GAM 77] developed a database design decision support system based on the DBTG model [COD 71]. Earlier, similar work on the design support system for network model databases appeared in [MIT 75] and [DE 78].

In most past research a common problem is that an optimal solution can be found only by exhaustively searching through all possible access configurations. The number of possible access configurations, however, can be intolerably large even when a small database is considered. In an effort to accomplish automatic design of physical database without an exhaustive search, Schkolnick and Tiberio [SCH 79] developed an algorithm based on partial exhaustive search. A certain number of intermediate solutions that are best at any design stage are saved, and an exhaustive search is performed starting from those intermediate solutions to a predefined depth in the search tree. The same number of best solutions in the results are saved and the procedure is repeated. A physical database design aid system (DBDSGN) [FIN 82] for system R has been implemented using this algorithm. One interesting feature of the system is that the algorithm uses System R's own optimizer as the cost evaluator. The validity of the heuristic involved, however, has not been well established.

Although this algorithm significantly improves the time complexity compared with the exhaustive-search approach, it still has a potential of being excessively time consuming when a very large database is designed. Certainly, a more efficient algorithm needs to be developed. In subsequent chapters in this dissertation we shall develop a formal method of partitioning the design problem into disjoint subproblems in order to reduce the time complexity. We then develop physical database design algorithms based on this formal method. Heuristics are subsequently employed to further reduce the time complexity.

1.1.4 Index selection

The index selection problem is an interesting subproblem of the access structure selection problem. The problem is concerned with finding an optimal set of indexes that minimizes the total transaction-processing cost. There has been a significant research effort on this problem. A pioneering work based on a simple cost model appeared in [LUM 71]. Some approaches [KIN 74], [STO 74] attempted to formalize the problem in order to find analytic results in certain restricted cases. In a more theoretical approach Comer [COM 78] proved that even a simplified version of the index selection problem is NP-complete. Thus, the best known algorithm to find an optimal solution would have an exponential time complexity. In an effort to find a more efficient algorithm, Schkolnick [SCH 75] discovered that, if the cost function satisfies a property called *regularity*, the complexity of the optimal index selection can be reduced to less than exponential. Hammer and Chan [HAM 76] took a somewhat different approach and developed a heuristic algorithm that significantly reduced the time complexity.

Most previous approaches towards optimal index selection, however, are limited to single-file cases. Furthermore, they only deal with secondary indexes without considering indexes coupled to the primary structure (clustering) of the file. Solutions for multifile cases or for the cases in which the primary structure is incorporated will be presented in Chapter 5.

1.2 Objective of the Dissertation

In this dissertation we concentrate on the access structure selection problem among the issues on physical database design surveyed in Section 1.1. In addition, the index selection problem will be studied as a subproblem of the access structure selection problem. In other words, we consider the problem of selecting the optimal access configuration of a database using the access structures that a particular DBMS we have at hand provides for the physical database design. The file modelling problem will not be explicitly considered; but, the techniques for solving this problem could help the implementation of the access structures themselves which we assume are already available. Hence, from now on, we consider the physical database design as a synonym for the access structure selection.

Most of previous research on physical database design concentrated on developing a cost evaluator, and selection of optimal access configuration remained dependent on the designer's intuition or an exhaustive search through all possible access configurations. Although an exhaustive search guarantees finding an optimal solution, it is practically impossible even with a small-sized database. This point is illustrated in Example 1.1.

Example 1.1: We look into a very simplified design process of a small database based on an exhaustive-search algorithm. We assume that the only access structure available is the clustering property. A column is said to have the *clustering property*, if a relation is stored according to the order of the column values. Although the clustering property can be assigned to a combination of multiple columns, in this example, we assume for simplicity that it can be assigned only to a single column.

Using this access structure, for a given set of transactions as input information, we want to find an optimal access configuration for the database consisting of relations R_1 and R_2 each of which owns two attributes. We have nine possible access configurations as in Figure 1-1, in which dashed lines show the position of the clustering column.

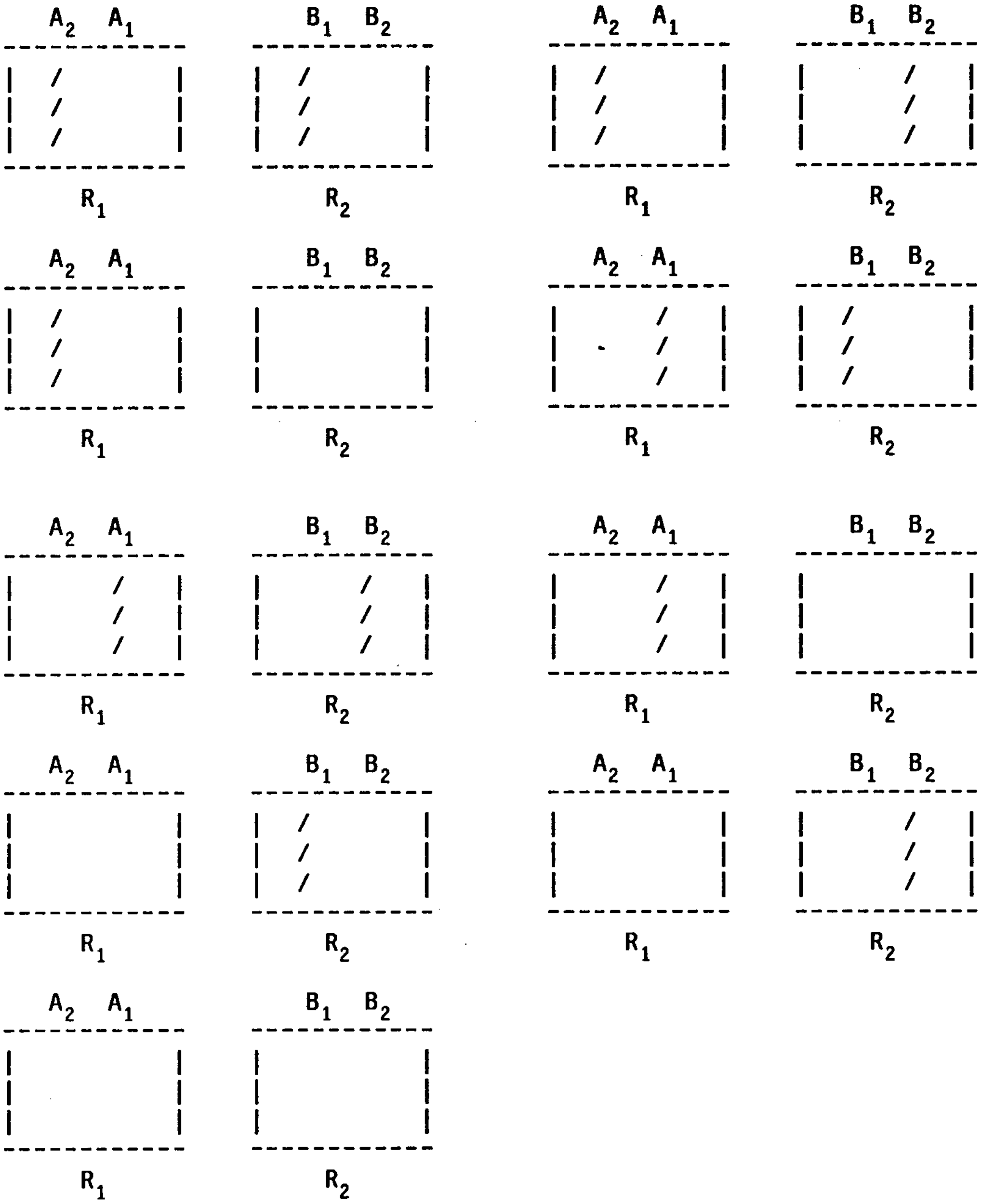


Figure 1-1: Nine Access Configurations.

The optimal access configuration can bc found as follows:

```
1. for each of the nine configurations:
    1.1. find the best join method for each query
    1.2. obtain the total processing cost
2. select the configuration that yields the minimum processing cost
```

In this simple design example, we have only nine possible access configurations; but the number of access configurations is explosive if we have more relations, more attributes in a relation, and various kinds of access structures such as the clustering property, indexes, links, etc. For instance, if we have five relations having five attributes each, with indexes and the clustering property as available access structures, the number of possible access configurations becomes

$$(6\times6\times6\times6\times6)\times(2^5\times2^5\times2^5\times2^5\times2^5) = 2.6\times10^{11} \ \square$$

As we see in Example 1, the cost of the exhaustive-search method becomes intolerably high even with a very small database. As pointed out in [GER 77], a relevant partitioning of the entire design is necessary to make the optimal design of physical databases a practical matter.

In this dissertation we shall develop a methodology for the design of multifile physical databases so that it can be applied to many situations with reasonable efficiency and accuracy. In particular, we discuss the issues involved in designing the access configuration of a physical database so as to minimize the total processing cost of input transactions – including queries and update transactions. In calculating the processing cost we only consider the number of I/O accesses; the cost due to the CPU time is not included. Our approach is somewhat formal and mathematical, deliberately avoiding excessive reliance on heuristics. Our purpose is to render the whole design phase manageable and to facilitate understanding of underlying mechanisms.

We proceed by first developing a design theory called *separability* that enables us to partition the entire design problem into disjoint subproblems. We then show that important subsets of features provided by both relational and network model database systems satisfy the conditions for separability. Thus, if features are restricted to these subsets, the optimal design of access configurations of multifile databases can be reduced to the collective optimal designs of smaller objects. In principle, these smaller objects, which we call *logical objects*, can be any subsets of the database being designed. However, in practice, the most convenient partition would be the set of individual relations for relational systems or record types for network model database systems. According to the theory, a basic design is obtained by using only features that satisfy the conditions for separability. This basic design is then extended, using some straightforward heuristics, to include other features provided by database management systems.

In Chapter 2 we develop the skeleton of the theory of separability and prove the separability theorem. We then investigate, in Chapter 3, how the theory can be applied to relational database systems. The application of the theory to network model database systems is presented in Chapter 8. Physical database design algorithms for multifile relational databases that are based on the theory and extended by heuristics are presented in Chapter 4. The index selection problem is an important subproblem of the physical database design problem. For this reason it is given a separate consideration in Chapter 5. The algorithms developed in Chapter 4 are fully implemented in 6000 lines of Pascal code. The cost formulas used in the implementation are summarized in Chapter 6. In developing cost formulas the function that estimates the number of block accesses when randomly selected tuples are retrieved in their physical order plays a particularly important role. The exact form of this function and various approximation formulas for faster evaluation are summarized in Chapter 7. Finally, briefly discussed in Chapter 9 are extensions of the design algorithms to the transactions that involve more-than-two variables.

2. Theory of Separability

The complexity of the physical database design stems from the interaction among the individual logical objects in the process of physical design. This interaction among logical objects prevents us from designing the access configurations of individual logical objects independently of one another because the optimal access configuration of a logical object is dependent on the access configurations of other logical objects.

A major cause of this interaction, in turn, is the join operation in relational databases or the SET traversal in network model databases (we shall simply call these two operations as the join operation). The cost of a join operation depends on access configurations of all logical objects participating in the join. Accordingly, we cannot determine the optimal access configuration of a particular logical object without the knowledge on the optimal access configurations of other logical objects. Similarly, the optimal configurations of other logical objects may depend on this particular logical object. Thus, we conclude that, in the most general cases, the only possible approach is to design the optimal configurations of all the logical objects simultaneously. But, as shown in Example 1.1, the complexity of this approach is intolerable.

However, we shall show in this chapter that, given a certain set of restrictions, the problem of optimally designing the access configuration of the entire database can be reduced to the subproblems of optimizing individual logical objects in the database independently of one another. The theorem of separability presented below formalizes this idea. Before introducing the theorem we need the following definitions.

Definition 2.1: The procedure of designing the optimal access configuration of a database is *separable* if it can be decomposed into the tasks of designing the optimal configurations of individual logical objects independently of one another. □

Definition 2.2: A *partial-operation cost* of a transaction is the part of the transaction processing cost that corresponds to the access to only one logical object, as well as of the auxiliary access structures defined for it. □

Definition 2.3: A *partial operation* is a conceptual division of the transaction whose processing cost is a partial-operation cost. □

Theorem 2.1: **(Separability)** The procedure of designing the optimal access configuration of a database is *separable* if the following conditions are satisfied:

1. The partial-operation cost of a transaction for a logical object is independent of both the access configuration specified and the partial operations used for the other logical objects.
2. A partial operation for a logical object can be chosen regardless of the access configuration specified and the partial operations used for the other logical objects.
3. Access structures for a logical object can be chosen independently of access configurations of the other logical objects.

Proof: Condition 2 states that, in selecting a partial operation of a transaction for a logical object, we are constrained neither by the access configurations of the other logical objects nor due to the partial operations used for them. Similarly, Condition 3 says that we are free to choose any access structures for a logical object regardless of the access structures chosen for the other logical objects. Furthermore, from Condition 1, a partial-operation cost of a transaction for a particular logical object, given a specific access configuration of the logical object, is affected neither by the access configurations of the other logical objects nor due to the partial operations used for them. Therefore, the partial operation cost of a transaction for a logical object is in no way affected by design decisions—choices of access structures and partial operations—of the other logical objects; nor do design decisions of a logical object affect the partial operation costs of transactions for the other logical objects. Thus, we can design individual logical objects independently of one another. Q.E.D.

Many database management systems satisfy Condition 3 in the sense that they do not put any restrictions in assigning access structures to different logical objects so that we can choose any access structures for a logical object regardless of the access structures assigned to other logical objects. Therefore, from now on, we exclude Condition 3 from our consideration.

Condition 1 is easy to check if we have specific cost formulas, but is somewhat difficult otherwise. In this case we define the following conditions which are sufficient and easier to check.

Sufficient conditions for Condition 1: The three items below are independent of the access configurations specified for the other logical objects and the partial operations used for the other logical objects.

1.1. Cardinality of the set of records accessed in the partial operation

1.2. The order according to which these records are accessed

1.3. Relative placement of these records in the storage medium

The partial operation cost of a transaction for a logical object, which represents the cost of accessing the set of records selected for this logical object, can be determined from these three items because they specify the number of records to be accessed, the locations of the records in the storage medium and the order of accessing those records. Thus, Conditions 1.1, 1.2, and 1.3 together form a sufficient condition for Condition 1 in Theorem 2.1 since they state that the three items in a logical object, and accordingly the partial operation cost of a transaction, are independent of the design decisions for the other logical objects. Note that these conditions are not necessary conditions because, although very unlikely, partial operation costs could be the same even though one the conditions is not satisfied.

We have now stated the conditions for separability in Theorem 2.1. Since Condition 3 is usually satisfied by database management systems, we consider only Conditions 1 and 2 in subsequent chapters. Three sufficient conditions for Condition 1 for separability have been presented.

Condition 1 will be substituted by these sufficient conditions whenever specific cost formulas are not available.

3. Separability in Relational Database Systems

3.1 Introduction

In this chapter and Appendix A we investigate how the theory outlined in Chapter 2 can be applied to relational database systems. Appendix A is a preliminary version of this chapter as is published in the Proceedings of the Seventh International Conference on Very Large Databases held in Cannes, France, in September 1981. We shall prove that a set of join methods which are important in practice satisfies the conditions for separability. The implication is that, if the available join methods are restricted to this set, the optimal design of the access configuration of a multifile database can be reduced to the collective optimal designs of individual relations. The physical designs thus obtained will be extended, using some straightforward heuristics, to incorporate other join methods as well. This extension will be discussed in Chapter 4.

Section 3.2 introduces major assumptions, while Section 3.3 describes applicable join methods of interest. In Section 3.5 we analyze those join methods and proves that an important subset has the separability property. We first proceed by presenting a series of case analyses using the simple cost model introduced in Section 3.4 and defining necessary terms. The ideas thus obtained are summarized in Subsection 3.5.3.

3.2 Approaches and Assumptions

We assume that the DBMS we consider provides as access structures indexes and the clustering property of a single relation. Clustering of two or more relations, as is supported in many hierarchical organizations, is not considered.

The database is assumed to reside on disklike devices. Physical storage space for the database is divided into units of fixed size called blocks [WIE 83]. The block is not only the unit of disk allocation, but is also the unit of transfer between main memory and disk. We assume that a block that contains tuples of a relation contains only tuples of that relation. Furthermore, we assume that the blocks containing tuples of a relation can be accessed serially. However, the blocks do not have to be contiguous on the disk.[1] For simplicity, we assume that a relation is mapped into a single file. Accordingly, from now on, we shall use the terms *file* and *relation* interchangeably; nor shall we make any distinction between an attribute and a column or between a tuple and a record.

We shall develop a simple cost model of the storage structure in Section 3.4, and shall use various cost formulas based on this model for case studies. We assume that no block access will be incurred if the next tuple (or index entry) to be accessed resides in the same block as that of the current tuple (or index entry); otherwise, a new block access is necessary. We also assume that all TID (tuple identifier) manipulations can be performed in main memory without any need for I/O accesses.

We consider only one-to-many (including one-to-one) relationships between relations. It is argued in Appendix G that many-to-many relationships between relations are less important for the optimization purpose. Note that here we are dealing with relationships in relational representations based on the equality of join-attribute values; a many-to-many relationship among distinct entity sets at the conceptual level is often structured with an additional intermediate relation [ELM 80].

Finally, we are considering only one-variable or two-variable queries in this chapter. For a query of more than two variables, a heuristic approach can be employed to decompose it into a sequence of two-variable queries (These correspond to one-overlapping queries in [WON 76]). The decomposition approach will be discussed in Chapter 9.

[1] For example, blocks of a file can be spread over the disk while they are connected as a linked list or linked implicitly by a file map.

3.3 Transaction Evaluation

3.3.1 Queries

The class of queries we consider is shown in Figure 3-1. The conceptual meaning of this class of queries is as follows. Tuples in relation R_1 are restricted by restriction predicate P_1. Similarly, tuples in relation R_2 are restricted by predicate P_2. The resulting tuples from each relation are joined according to the join predicate $R_1.A = R_2.B$, and the result projected over the columns specified by <list of attributes>. We call the columns that are involved in the restriction predicates *restriction columns*, and those in the join predicate *join columns*. The actual implementation of this class of queries does not have to follow the order specified above as long as it produces the same result.

$$\begin{array}{lll} \text{SELECT} & \langle\text{list of attributes}\rangle & \\ \text{FROM} & R_1, R_2 & \\ \text{WHERE} & R_1.A = R_2.B & \text{AND} \\ & P_1 & \text{AND} \\ & P_2 & \end{array}$$

Figure 3-1: General Class of Queries Considered.

Query evaluation algorithms, especially for two-variable queries, have been studied in [BLA 76] and [YAO 79]. The algorithms for evaluating queries differ significantly in the way they use join methods. Before discussing the various join methods, let us define some terminology. Given a query, an index is called a *join index* if it is defined for the join column of a relation. Likewise, an index is called a *restriction index* if it is defined for a restriction column. We use the term *subtuple* for a tuple that has been projected over some columns. The restriction predicate in a query for each relation is decomposed into the form Q_1 AND Q_2, where Q_1 is a predicate that can be processed by using indexes, while Q_2 cannot. Q_2 must be resolved by accessing individual records. We shall call Q_1 the *index-processible predicate* and Q_2 the *residual predicate.*

Some algorithms for processing joins that are of practical importance are summarized below (see also [BLA 76] [SEL 79]):

- *Join Index Method*: This method presupposes the existence of join indexes. For each relation, the TIDs of tuples that satisfy the *index processible predicates* are obtained by manipulating the TIDs from each index involved; the resultant TIDs are stored in temporary relations R_1' and R_2'. TID pairs with the same join column values are found by scanning the join column indexes according to the order of the join column values. As they are found, each TID pair (TID_1, TID_2) is checked to determine whether TID_1 is present in R_1' and TID_2 in R_2'. If they are, the corresponding tuple in one relation, say R_1, is retrieved. When this tuple satisfies the *residual predicate* for R_1, the corresponding tuple in the other relation R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple of interest is constructed. (We say that the direction of the join is from R_1 to R_2.)

- *Sort-Merge Method*: The relations R_1 and R_2 are scanned—either by using restriction indexes, if there is an index-processible predicate in the query, or by scanning the relation directly. Restrictions, partial projections, and the initial step of sorting are performed while the relations are being initially scanned and stored in temporary relations T_1 and T_2. T_1 and T_2 are sorted by the join column values. The resulting relations are scanned in parallel and the join is completed by merging matching tuples.

- *Combination of the Join Index Method and the Sort-Merge Method*: One relation, say R1, is sorted as in the sort-merge method and stored in T_1. Relation R_2 is processed as in the join index method, storing the TIDs of the tuples that satisfy the index processible predicates in R_2'. T_1 and the join column index of R_2 are scanned according to the join column values. As matching join column values are found, each TID from the join index of R_2 is checked against R_2'. If it is in R_2', the corresponding tuple in R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple is constructed.

- *Inner/Outer-Loop Join Method*: In the two join methods described above, the join is performed by scanning relations in the order of the join column values. In the inner/outer-loop join, one of the relations, say R_1, is scanned without regard to order, either by using restriction indexes or by scanning the relation directly. For each tuple of R_1 that satisfies predicate P_1, all tuples of relation R_2 that satisfy predicate P_2 and the join predicate are retrieved and concatenated with the tuple of R_1. The subtuples of interest are then projected upon the result. (We say the direction of the join is from R_1 to R_2.)

Let us note that, in the combination of the join index method and the sort-merge method, the operation performed on either relation is identical to that performed on one relation in the join index method or in the sort-merge method. We call the operations performed on each relation *join index method (partial)* or *sort-merge method (partial)*, respectively; whenever no confusion arises, we call these operations simply *join index method* or *sort-merge method.* According to these definitions, the complete join index method actually consists of two join index methods (partial) and, similarly, the complete sort-merge method consists of two sort-merge methods (partial).

3.3.2 Update transactions

We assume that the updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Thus, updates are not performed on the join of two or more relations. (If they are, certain ambiguity arises on which relations to update [KEL 81].) The class of update transactions we consider is shown in Figure 3-2.

$$\begin{array}{lll} \text{UPDATE} & R_1 & \\ \text{SET} & R_1.C = \langle\text{new value}\rangle & \\ \text{FROM} & R_1, R_2 & \\ \text{WHERE} & R_1.A = R_2.B & \text{AND} \\ & P_1 & \text{AND} \\ & P_2 & \end{array}$$

Figure 3-2: General Class of Update Transactions Considered.

The conceptual meaning of this class of transactions is as follows. Tuples in relation R_2 are restricted by restriction predicate P_2. Let us call the set of resulting tuples T_2. Then, the value for column C of each tuple in R_1 is changed to <new value> if the tuple satisfies the restriction predicate P_1 and has a matching tuple in T_2 according to the join predicate. In a more familiar syntax [CHA 76], the class of update transactions can be represented as in Figure 3-3. The equivalence of the two representations has been shown for queries in [KIM 82].

Deletion transactions are specified in an analogous way. It is assumed that insertion transactions refer only to single relations. From now on, unless any confusion arises, we shall refer to update, deletion or insertion transactions simply as update transactions.

The update transaction in Figure 3-2 can be processed just like queries except that an update

```
UPDATE R1
SET    R1.C = <new value>
WHERE  P1      AND
       R1.A    IN
        (SELECT R2.B
         FROM   R2
         WHERE  P2 )
```

Figure 3-3: An Equivalent Form of the General Class of Update Transactions.

operation is performed instead of concatenating and projecting out the subtuples after relevant tuples are identified. In particular, all the join methods described in Section 3.3.1 can be used for update transactions as well to resolve the join predicates (ones that relate the two relation) that they have. But, there are two constraints: 1) The sort-merge method cannot be used for the relation to be updated since it is meaningless to create a temporary sorted file to update the original relation. 2) When the inner/outer-loop join method is used, the direction of the join must be from the relation to be updated (R_1) to the other relation (R_2) because, if the direction were reversed, the same tuple might be updated more than once.

3.4 Cost Model of the Storage Structure

To calculate the cost of evaluating a query, we need a proper model of the underlying storage structure and its corresponding cost formula. Although the theory does not depend on the specifics of cost models, it is helpful to have a simple cost model for illustrative purposes.

We assume that a B^+-tree index [BAY 72] can be defined for a column or for a set of columns of a relation. The leaf-level of the index consists of pairs (key and TID) for every tuple in that relation. The leaf-level blocks are chained according to the order of indexed column values, so that the index can be scanned without traversing the index tree. Entries having the same key value are ordered by TID.

An index is called a *clustering index* if the relation for which this index is defined is physically clustered according to the index column values. With a clustering index, we assume that no block is fetched more than once when tuples with consecutive values of the indexed column are retrieved. Except for this ordering property, no other difference in the structure is assumed between a clustering and a nonclustering index. The clustering property can greatly reduce the access cost. Unfortunately, only one column of a relation can have the clustering property, since clustering requires a specific order of records in the physical file. One of the objectives of designing optimal physical databases is to determine which column will be assigned the clustering property.

The access cost will be measured in terms of the number of I/O accesses. The following notation will be used throughout this chapter:

R	: A relation.
Other(R)	: The relation to be joined with R.
C	: A column.
n_R	: Number of tuples in relation R (cardinality).
p_R	: Blocking factor of relation R.
L_C	: Blocking factor of the index for column C.
F_C	: Selectivity of column C or its index
cc	: Subscript for the clustering column.
m_R	: Number of blocks in relation R, which is equal to n_R/p_R.
im_C	: Number of blocks that the index for column C occupies.
t	: A transaction
$H_{t,R}$	: Projection factor of transaction t on relation R.

By using the simplified model above, the cost of various operations can be obtained as follows:

- Relation Scan Cost – Cost for serially accessing all the blocks containing the tuples of a relation:

$$RS(R) = n_R/p_R = m_R$$

- Index Scan Cost – Cost for serially accessing the leaf- level blocks of an entire index:

$$IS(I,R) = \lceil n_R/L_C \rceil$$

- Index Access Cost – Cost for one access of the index tree from the root:

$$IA(I,R) = \lceil \log_{L_C} n_R \rceil + \lceil F_C \times n_R/L_C \rceil$$

- Sorting Cost – Cost for sorting a relation, or a part thereof, according to the values of the columns of interest:

$$SORT(NB) = 2 \times \lceil NB \rceil + 2 \times \lceil NB \rceil \times \lceil \log_z \lceil NB \rceil \rceil$$

Here we assume that a z-way sort-merge is used for the external sort [KNU-b 73]. NB is the number of blocks in the temporary relation containing the subtuples to be sorted after restriction and projection have been resolved. It will be noted that SORT(NB) does not include the initial scanning time to bring in the original relation, while it does include the time to scan the temporary relation for the actual join after sorting (see [BLA 76]).

3.5 Design Theory

In this section we investigate the property of separability for relational database systems. In particular, we shall prove that the set of join methods consisting of the join index method, the sort-merge method, and the combination of the two satisfies the conditions for separability under certain constraints. The inner/outer-loop join method is a nonseparable join method with respect to this separable set. The design algorithms will be extended to incorporate this join method in Chapter 4. We facilitate comprehension through a series of examples and by case analysis, using the cost model developed in Section 3.4. Observations resulting from this procedure are formalized and proved in Section 3.5.3.

Our approach to physical database design is based on the premise that at execution time the query processor will choose the best processing method for a given query. We call this processor an *optimizer.* Since the behavior of the optimizer at execution time affects the physical database design critically, we investigate this issue and discuss how it is related to the design.

We define the influence of the restriction on one relation to the number of tuples to be retrieved in the other relation participating in a join as the *coupling effect* (which is similar in concept to the *feedback* mentioned in [YAO 79]). Starting with a case in which coupling effects between relations are not considered, we then proceed to those cases in which they are included.

3.5.1 Cases without coupling effects

Example 3.1: Figure 3-4 describes two relations R_1 and R_2 with their access configurations. Dashed lines (/) represent clustering indexes, the dotted lines (:) nonclustering indexes. Columns without either type of line have no indexes defined for them. We would like to find the best method of evaluation – which the optimizer would choose at query-processing time, for the following query:

SELECT A_1, A_2, B_2
FROM R_1, R_2
WHERE $R_1.A_2 =$ 'a_2' AND
$R_2.B_2 =$ 'b_2' AND
$R_1.A_1 = R_2.B_1$

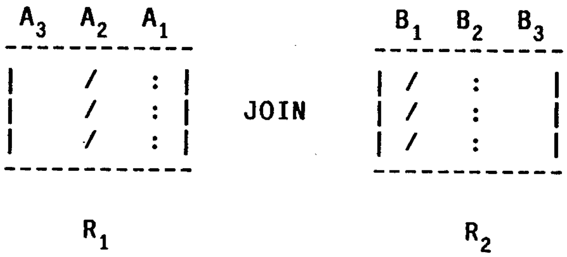


Figure 3-4: Relations R_1 and R_2.

For this example only, it is also assumed that all the tuples in each relation participate in the join.

Given these assumptions, the optimizer could try all the possible combinations of the join methods, evaluate the cost of each, and then select the one that costs the least. We have here the following combinations:

R_1		R_2
1. Join index method (partial)	and	Join index method (partial)
2. Sort-merge method (partial)	and	Sort-merge method (partial)
3. Join index method (partial)	and	Sort-merge method (partial)
4. Sort-merge method (partial)	and	Join index method (partial)

Using the cost model given in Section 3.4, the following formulas give the cost (number of block accesses) for each of the four cases above. In each formula the first and second bracketed expressions represent the cost of accessing relations R_1, and R_2 respectively. Bracketed expressions in the formulas are given arbitrary values for illustrative purposes. Those expressions whose form is identical are given the same value.

$$\begin{aligned} \text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) + F_{A2} \times n_{R1}] + && :100 + \\ & [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})] && :20 \end{aligned} \qquad (3.1)$$

$$\begin{aligned} \text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} + SORT(F_{A2} \times H_{R1} \times m_{R1})] + && :60 + \\ & [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) + SORT(F_{B2} \times H_{R2} \times m_{R2})] && :50 \end{aligned} \qquad (3.2)$$

$$\begin{aligned} \text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) + F_{A2} \times n_{R1}] + && :100 + \\ & [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) + SORT(F_{B2} \times H_{R2} \times m_{R2})] && :50 \end{aligned} \qquad (3.3)$$

$$\begin{aligned} \text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} + SORT(F_{A2} \times H_{R1} \times m_{R1})] + && :60 + \\ & [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})] && :20 \end{aligned} \qquad (3.4)$$

Here b(m,p,k) is a function that provides the number of block accesses, where k is the number of tuples to be retrieved in the order of TID values (TID order). An exact form of this function and various approximation formulas are summarized in Chapter 7. The function is approximately linear in k when $k \ll n$, and approaches m asymtotically as k becomes large. A simple approximation suggested by Cardenas [CAR 75] is $b(m,p,k) = m\,[1 - (1-1/p)^k]$. F_{A2} and F_{B2} are the selectivities of the columns $R_1.A_2$ and $R_2.B_2$, respectively. In Equation (3.1), $F_{A2} \times n_{R1}$ and $b(m_{R2},p_{R2},F_{B2} \times n_{R2})$ represent the numbers of blocks accessed that contain data tuples of relations R_1 and R_2, respectively. Since retrieving tuples by scanning a nonclustering join index will access the tuples randomly, the same block will be accessed repeatedly if it contains more than one tuple. Therefore, one block access is needed to retrieve each tuple. Hence we get $F_{A2} \times n_{R1}$ for the number of data blocks fetched from relation R_1. On the other hand, for relation R_2, the join index is clustering and thus the tuples will be retrieved in TID order. Therefore, even though a block contains more than one tuple, each block will be fetched only once. We thus get $b(m_{R2},p_{R2},F_{B2} \times n_{R2})$ for the number of data blocks fetched from R_2, where $F_{B2} \times n_{R2}$ is the number of tuples selected by the restriction.

In Equation (3.2), $F_{A2} \times m_{R1}$ and $b(m_{R2},p_{R2},F_{B2} \times n_{R2})$ represent the numbers of blocks accessed during the initial scan of the relation prior to sorting. Since the restriction index is clustering in relation R_1, the initial scan through this restriction index will access $F_{A2} \times m_{R1}$ blocks. In relation R_2, a nonclustering restriction index is used to access the relation initially. This restriction results in random distribution of TIDs of the qualified tuples over the blocks. Since these tuples are then accessed in TID order, the access cost is $b(m_{R2},p_{R2},F_{B2} \times n_{R2})$.

The factor H_{R2} used in the Equation (3.3) represents the projection effect upon relation R_2. Since the projection selects only part of the attributes from the relations, the tuple is usually smaller after projection. The cost of writing the final result is not included since it is the same regardless of the join method used.

With the specific values of the access cost given, Equation (3.4) gives the minimum access cost. We note that the access costs for each relation do not depend on any parameter of any other relation, and that each part of the cost of Equation (3.4) becomes the local minimum. That is, the first part of the cost incurred by accessing relation R_1 is the minimum of the costs of the join methods used for R_1, while the second part is the minimum of those for R_2. This implies that the optimizer can determine the optimal join method on one relation without regard to any properties of other relations. [END Example 3.1]

The foregoing observation is extremely important because, if we can determine the optimal join method for one relation without regard to other relations, we can also determine the optimal access configuration for the relation without regard to other relations using the following procedure:

1. Consider each possible access configuration for a relation in turn.

2. Find the best join method of each transaction given the particular access configuration.

3. Calculate the total cost for processing the transactions, using their expected frequency of occurrence.

4. Repeat this procedure for all other possible access configurations, finally selecting the one that yields the minimal total cost.

The result of this will be to reduce the problem of designing an optimal access configuration of a database to the problem of designing access configurations of single relations. Therefore, local optimal solutions for individual relations constitute an optimal solution for the entire database.

In Example 3.1 we considered only the cases without coupling effects. It will be shown, in the following discussion, that the problem is similarly reduced even when coupling effects are actually present. Before further discussion, we need the following definition and example.

Definition 3.1: The *join selectivity* J(R,JP) of a relation R with respect to a join path JP is the ratio of the number of distinct join column values of the tuples participating in the unconditional join to the total number of the distinct join column values of R. A *join path* is a set (R_1,R_1.A,R_2,R_2.B), where R_1 and R_2 are relations participating in the join and R_1.A and R_2.B are the join columns of R_1 and R_2, respectively. An *unconditional join* is a join in which the restrictions on either relation are not considered. □

Definition 3.2: A *connection* is a join path predefined in the schema [WIE 79]. □

Join selectivity is the same as the ratio of the number of tuples participating in the unconditional join to the total number of tuples in the relation (cardinality of the relation). Join selectivity is generally different in R_1 and R_2 with respect to a join path, as shown in the following example:

Example 3.2: Let us assume that the two relations in Figure 3-5 have an 1-to-N partial-dependency relationship. Partial dependency means that every tuple in the relation R_2 that is on the N-side of the relationship has a corresponding tuple in R_1, but not vice versa [ELM 80]. Let us assume that 50% of the countries have at least one ship so that the tuples representing those countries participate in the unconditional join. Every tuple in the SHIPS relation (R_2) participates in the unconditional join according to the partial dependency. The join selectivity of the COUNTRIES relation is then 0.5, while that of the SHIPS relation is 1.0. □

R_1	COUNTRIES(Countryname, Population)
R_2	SHIPS(ShipId, Country, Crewsize, Deadweight)

Figure 3-5: COUNTRIES and SHIPS Relations.

3.5.2 Cases with coupling effects

Let us investigate the four cases shown in Example 3.1 – using the same query, join methods, and access configuration defined as in Figure 3-4, but now with coupling effects. In fact, we shall consider coupling effects throughout our subsequent discussions. We shall also assume that R_1 and R_2 have a 1-to-N relationship (1 for R_1 and N for R_2).

Case 1: The join index method is applied to both relations R_1 and R_2. With coupling effect, the join will be performed as follows: If a tuple of relation R_1 does not satisfy the restriction predicate for R_1, the corresponding tuples of R_2 that have the same join column values are not accessed. Hence, we have the coupling effect from R_1 to R_2. If there are only index-processible predicates in the query to be evaluated, the situation is then symmetric – in the sense that, for the tuples in relation R_2 that do not satisfy the restriction predicate for R_2, the corresponding tuples of R_1 are not accessed either. We have this symmetry because we can resolve all index-processible predicates by using TIDs only, without any need to access the data tuples themselves.

Since both $R_1.A_2$ and $R_2.B_2$ have indexes defined for them, the restriction predicates in the WHERE clause are index-processible. Therefore, the cost of evaluating this query, including the coupling effect, will be as follows:

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2},R_1) + IS(I_{A1},R_1) + \{\langle J_1 \times b(1/F_{B1},F_{B1} \times n_{R2}, \\ & F_{B2} \times n_{R2})/(1/F_{B1})\rangle \times F_{A2} \times n_{R1}\}] + \\ & [IA(I_{B2},R_2) + IS(I_{B1},R_2) + b(m_{R2},p_{R2},\{\langle J_2 \times F_{A2}\rangle \times F_{B2} \times n_{R2}\})]\end{aligned}$$

Here J_1 and J_2 represent the join selectivity of relations R_1 and R_2, respectively, for the join path considered. Exprcssions in the braces represent the numbers of data tuples accessed in relations R_1

and R_2, respectively. In the first part of the formula, the expression in the braces simultaneously represents the number of blocks accessed in relation R_1. This follows the argument shown in Example 3.1.

F_{B1} is the selectivity of column $R_2.B_1$ and $1/F_{B1}$ represents the number of groups[2] of tuples that have the same join column values in relation R_2 – which is essentially the same as the number of distinct join column values.

The expression $b(1/F_{B1},F_{B1} \times n_{R2}, F_{B2} \times n_{R2})$ represents the number of groups selected by restriction F_{B2}. Although the b function estimates the number of block accesses in which a certain number of tuples are randomly selected, the same function is used for estimating the number of logical groups selected – if the latter are assumed to be of uniform size. Note that the clustering or nonclustering of tuples in a group is irrelevant. The product $F_{B1} \times n_{R2}$, the number of tuples in one logical group, plays a role similar to that of the blocking factor.

The expression $b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2})/(1/F_{B1})$ represents the ratio of the number of groups selected by restriction F_{B2} to the total number of groups in relation R_2. Since every tuple participating in the unconditional join in R_1 has a unique join column value and, accordingly, exactly one corresponding group in R_2 (let us recall that R_1 is on the 1-side of the 1-to-N relationship), this ratio correctly represents a special restriction upon R_1 caused by the coupling effect originating in R_2.

In the second part of the cost formula, we simply use F_{A2} to represent the coupling effect directed from R_1 to R_2. Since in R_1 every tuple has a unique join column value, if a tuple is selected according to the restriction, the corresponding group in R_2 that has the same join column value (if it exists) will be selected on the basis of this special restriction resulting from the coupling effect.

[2] *Group* here is very close in concept to *set occurrence* in CODASYL-type databases.

Hence, F_{A2} represents the ratio of the number of groups selected as a consequence of the coupling effect to the total number of groups in R_2 participating in the unconditional join. That ratio, in turn, has the same value as the ratio of tuples, selected according to the coupling effect, to the total number of tuples participating in the unconditional join in R_2. □

The coupling effect is formally defined as follows:

Definition 3.3: The *coupling effect* from relation R_1 to relation R_2, with respect to a transaction, is the ratio of the number of distinct join column values of the records of R_1, selected according to the restriction predicate for R_1, to the total number of distinct join column values in R_1. □

If we assume that the join column values are randomly selected, the coupling effect from R_1 to R_2 is the same as the ratio of the number of distinct join column values of R_2 selected by the effect of the restriction predicate for R_1 to the number of distinct join column values in R_2 participating in the unconditional join.

Definition 3.4: A *coupling factor* Cf_{12} from relation R_1 to relation R_2, with respect to a transaction, is the ratio of the number of distinct join column values of R_2, selected by both the coupling effect from R_1 (through the restriction predicate for R_1) and the join selectivity of R_2, to the total number of distinct join column values in R_2. □

According to the definition, a coupling factor can be obtained by multiplying the coupling effect from R_1 to R_2 by the join selectivity of R_2. This coupling factor contains all the consequences of the interactions of relations in the join operation, since it includes both coupling and join filtering effects. Let us note that, although the coupling factor can be obtained in any case, it does not always contribute to the reduction of the tuples to be retrieved. We will see an example of this in Case 2 below. A coupling factor is said to be *effective* if the coupling effect actually contributes to the reduction of the tuples to be retrieved. In Case 1, the expressions in angle brackets represent the coupling factors from R_2 to R_1 and from R_1 to R_2, respectively. Hence,

$$Cf_{12} = J_2 \times F_{A2},$$
$$Cf_{21} = J_1 \times b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2})/(1/F_{B1}).$$

One important observation here is that the coupling factors do not depend on the specific access structures present in either relation, nor on the specific join method selected, but rather (and solely) depend on the restriction and the data characteristics. Such characteristics include the side the relation is on in the 1-to-N relationship, the average number of tuples in one group, and the join selectivity – which will be known before we start the design phase.

Now let us investigate the remaining cases in which coupling effects are present between relations.

Case 2: The sort-merge join method is applied to both relations, in the same situation as in Figure 3-4. The cost formula is then as follows:

$$\begin{aligned}Cost = {} & [IA(I_{A2},R_1) + F_{A2} \times m_{R1} + SORT(F_{A2} \times H_{R1} \times m_{R1})] \\ & + [IA(I_{B2},R_2) + b(m_{R2},p_{R2},F_{B2} \times n_{R2}) + SORT(F_{B2} \times H_{R2} \times m_{R2})]\end{aligned}$$

It will be noted that the coupling factors do not appear in the cost formula. This is because, when the sort-merge join method is used, an initial scan and the sort are performed before the join is resolved; indexes are not used any more while the join is being actually resolved, since the relation scan is performed upon the sorted temporary relations. The coupling effect can arise only when the join is being actually resolved and only when the join index is used. Thus, the coupling factor is not effective in this case.

Case 3: The sort-merge join method is used for R_1, the join index method for R_2 – in the same situation as in Figure 3-4. The join will be performed as described in Section 3.3, under the heading "Combination of the Join Index Method and the Sort-Merge Method." Note that the coupling factor is effective from R_1 to R_2, but not from R_2 to R_1. Thus, we obtain the following cost formula:

$$\begin{aligned}Cost = {} & [IA(I_{A2},R_1) + F_{A2} \times m_{R1} + SORT(F_{A2} \times H_{R_1} \times m_{R1})] \\ & + [IA(I_{B2},R_2) + IS(I_{B1},R_2) + b(m_{R2},p_{R2},Cf_{12} \times F_{B2} \times n_{R2})]\end{aligned}$$

Case 4: The join index method is used on R_1, the sort-merge method on R_2–in the same situation as in Figure 3-4. We obtain the following cost formula:

$$\begin{aligned} \text{Cost} = {} & [IA(I_{A2},R_1) + IS(I_{A1},R_1) + Cf_{21} \times F_{A2} \times n_{R1}] \\ & + [IA(I_{B2},R_2) + b(m_{R2},p_{R2},F_{B2} \times n_{R2}) + SORT(F_{B2} \times H_{R2} \times m_{R2})] \end{aligned}$$

In all the cases above we note that the access cost for each relation is independent of any parameter of the other relation. Thus, when the optimizer chooses the least costly join methods, it can compare the costs for only one relation at a time.

3.5.3 Formalization

So far, we have discussed the property of separability for relational systems through a series of examples and case analyses. The ideas involved are now formalized. To begin with, we rephrase the definitions and the theorem presented in Chapter 2 to make them specifically suitable for relational systems.

Definition 3.5: The procedure of designing the optimal access configuration of a database is *separable* if it can be decomposed into the tasks of designing the optimal configurations of individual relations independently of one another. □

Definition 3.6: A *partial-join cost* is that part of the join cost that represents the accessing of only one relation, as well as the auxiliary structures defined for that relation. □

Definition 3.7: A *partial-join algorithm* is a conceptual division of the algorithm of a join method whose processing cost is a partial-join cost. □

Theorem 3.1: The procedure of designing the optimal access configuration of a database is *separable* if the following conditions are satisfied:

1. A partial-join cost for relation R can be determined regardless of the partial-join algorithm used and the access configuration defined for Other(R).

2. A partial-join algorithm can be chosen for R regardless of the partial-join algorithm used and the access configuration defined for Other(R). □

Additionally, we need the following definitions:

Definition 3.8: The *partial coupling effect* from relation R_1 to relation R_2, with respect to each transaction, is the ratio of the number of distinct join column values of the tuples of R_1, selected according to the index-processible predicate for R_1, to the total number of distinct join column values in R_1. □

Definition 3.9: A *partial coupling factor* PCf_{12} from relation R_1 to relation R_2 with respect to a transaction is the ratio of the number of distinct join column values of R_2, selected by both the partial coupling effect from R_1 (through the restriction predicate for R_1) and the join selectivity of R_2, to the total number of distinct join column values in R_2. □

Definition 3.10: The *restricted set* of relation R with respect to a transaction is the set of tuples of R selected according to the restriction predicate for R. □

Definition 3.11: The *partially restricted set* of relation R with respect to a transaction is the set of tuples of R selected according to the index-processible predicate for R. □

Definition 3.12: The *coupled set* of relation R_1 with respect to a transaction is the set of tuples in R_1 selected according to the coupling factor from R_2. □

Definition 3.13: The *partially coupled set* of relation R_1 with respect to a transaction is the set of tuples of R_1 selected according to the partial coupling factor from R_2. □

Definition 3.14: The *result set* of relation R with respect to a transaction is the intersection of the restricted set and the coupled set. Thus, the tuples in the result set satisfy all the predicates. □

Definition 3.10 to Definition 3.14 define various subsets of the relation according to the predicates they satisfy. In Figure 3-6 these subsets are graphically illustrated. Cardinalities of subsets of relation R_1 can be obtained as follows:

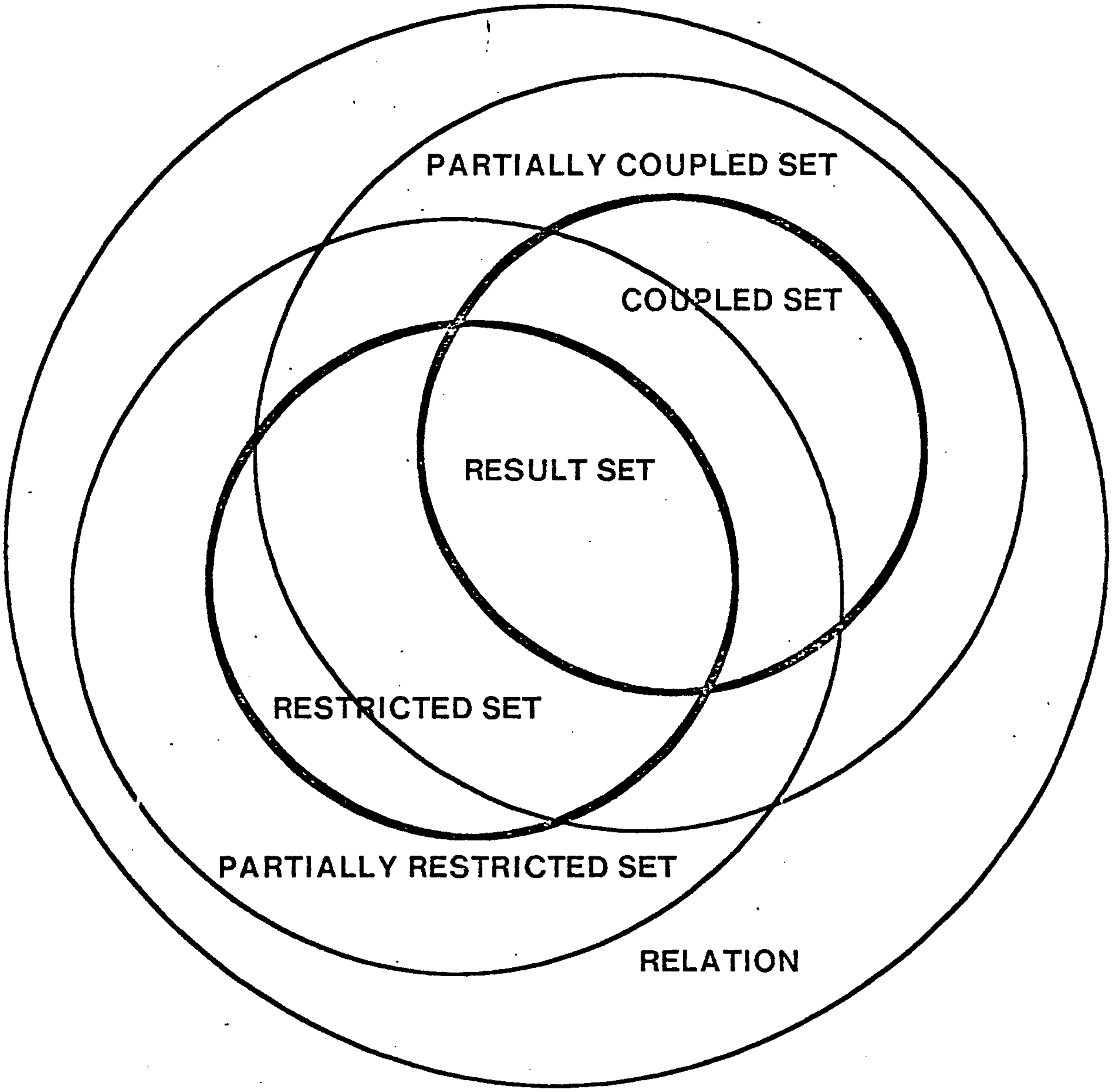


Figure 3-6: Various Subsets of a Relation.

|restricted set| $= n_{R_1} \times$ Selectivity of the restriction predicate

|partially restricted set| $= n_{R_1} \times$ Selectivity of the index-processible predicate

|coupled set| $= n_{R_1} \times Cf_{21}$

|partially coupled set| $= n_{R_1} \times PCf_{21}$

|result set| $= n_{R_1} \times Cf_{21} \times$ Selectivity of the restriction predicate

Using all the definitions and the theorem above, we prove the following theorem that shows the separability of a relational system.

Theorem 3.2: The set of join methods consisting of the join index method, the sort-merge method, and the combination method satisfies the conditions for separability under the constraint that, whenever the join index method is used for both relations, at least one relation must have indexes for all restriction columns.

Proof: In the set of join methods considered, there are two partial-join algorithms: the join index method (partial) and the sort-merge method (partial). Since these two can be arbitrarily combined to form a join method, Condition 2 for separability is satisfied. For Condition 1 of separability we prove that each of the three sufficient conditions is satisfied as follows:

Condition 1.1: We prove that the first condition is satisfied by showing that the following statements are true:

1. If the sort-merge method is used, the set of records in R that are accessed is the restricted/partially restricted set.
2. If the join index method is used, the set of records in R that are accessed is the intersection of restricted/partially restricted set and the coupled set.

Then, we know the set of records of R accessed is independent of the access structures of and the join methods used for Other(R) because the restricted/partially restricted set can be completely determined by local parameters of relation R, and the coupled set can be determined by the coupling effect and the join-filtering effect which are independent of the access structures of Other(R) and the partial-join algorithms used for Other(R).

Now, let us investigate each of the two statements. First, when the sort-merge method is used, it follows directly from the definition of this join method that the partially restrictive set will be accessed if there are residual predicates; the restricted set will be accessed otherwise. Second, when the join index method is used, the optimizer will access the indexes of the relation (say R) having indexes for all restriction columns first. Since the predicates for R are entirely resolved by using indexes, coupling factor is effective in Other(R). The data records of Other(R) will subsequently be accessed and the predicates for Other(R) are entirely resolved before accessing records of R. Thus, coupling factor is also effective in R. Since full – not partial – coupling factors are effective in both relations, the records to be accessed are in the coupled set. These tuples are also in the restricted/partially restricted set because the index-processible predicate is resolved by using indexes before data tuples are accessed.

Let us note that, if the optimizer accesses the indexes of Other(R) first, then only partial coupling factor is effective in R. But, because this will always cost more than the previous method, the optimizer will always choose the previous one.

Condition 1.2: The order of accessing those tuples is always the join column value order regardless of the access structures and partial-join algorithms used.

Condition 1.3: Since we assumed that a block contains tuples of only one relation, tuples of a relation cannot interfere with the placement of tuples of other relations. Q.E.D.

3.5.4 Separability in cases where arbitrary indexes are missing

The set of join methods in Theorem 3.2 does not have the separability property if, for any transaction, some restriction indexes are missing in both relations. Example 3.3 further illustrates this point.

Example 3.3: Let us assume that the join index method is used for both R_1 and R_2, in the same situation as in Figure 3-4, but that now restriction indexes for both R_1 and R_2 are missing. In this situation, since there are no restriction indexes, there is no way of resolving the restriction predicate without accessing the tuples themselves. Therefore, if we access relation R_1 first, the access cost would be

$$\text{Cost1} = [IS(I_{A1},R_1) + PCf_{21} \times n_{R1}] + [IS(I_{B1},R_2) + b(m_{R2},p_{R2},Cf_{12} \times n_{R2})]$$

On the other hand, if we access relation R_2 first, the access cost would then be

$$\text{Cost2} = [IS(I_{A1},R_1) + Cf_{21} \times n_{R1}] + [IS(I_{B1},R_2) + b(m_{R2},p_{R2},PCf_{12} \times n_{R2})]$$

Here, $PCf_{21}=J_1$ and $PCf_{12}=J_2$ since there are no restriction indexes in both R_1 and R_2. In general, if some restriction indexes are missing in both relations, the coupling factor is effective in one relation while the partial coupling factor is effective in the other relation. The choice depends on which relation is to be accessed first. The optimizer will choose the one that makes the join cost cheaper at run time based on the access configurations of both relations. Since this choice depends on the access configurations of both relations, the design is not separable.□

What's implied in the optimal design of the physical database is that those indexes that do not compensate for their maintenance and access costs should not be included in the result. Since Theorem 3.2 requires the existence of all the restriction indexes in at least one relation for each two-variable transaction, we can inevitably expect that, for some transactions, this constraint is not met during the decision process. In this situation calculation of the cost is no longer separable. Nevertheless, the error caused by the assumption of separability should not be significant because the restriction indexes for both relations that have been dropped must be relatively insignificant—otherwise, the indexes would not have been dropped.

3.5.5 Update cost

We assume here that updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Thus, updates are not performed on the join of two or more relations.

Imagine that the qualification part—which can be treated as a query— is segregated. Then, the remaining part—update operation—depends only on the local parameters of the relation to be updated and on the coupling factor because the update operation should only occur after all the predicates are resolved. When processing the qualification part, there are some restrictions as explained in Section 3.3.2. The restriction, however, is independent of the access structures or partial-join algorithms of other relations. Thus, separability can also be applied to the update transactions as well.

3.6 Summary

The theory of separability has been investigated in the context of relational database systems. In particular, it has been shown that the set of join methods consisting of the join index method, the sort-merge method, and the combination method has the property of separability. The implication is that, if the database system supports only this set of join methods, the physical database can be designed relation by relation independently of one another.

4. Physical Design Algorithms for Multifile Relational Databases

4.1 Introduction

In this chapter and Appendix D an algorithm for the optimal physical design of multifile databases will be presented. Appendix D contains detailed experimental data for the validation of the algorithm. This algorithm exploits the property of separability so that the entire design is partitioned into the designs of individual relations. The scheme is extended, using heuristics, to include the inner/outer-loop join method which cannot be incorporated by the theory of separability. The design of a single relation can still be a very complex problem. Thus, other heuristics are employed to further reduce the complexity of this design process.

In Section 4.2 the design algorithm is described in detail. Its time complexity is investigated in Section 4.3. Validation of the heuristics involved in the algorithm is briefly explained in Section 4.4.

4.2 Design Algorithm

The design algorithm is schematically illustrated in Figure 4-1.

The input information for and the output results from the design algorithm are described below:

Input:

- Usage information: A set of various queries and update transactions with their frequencies.
- Data Characteristics: The logical schema including connections; (for each relation in the database) cardinality, blocking factor, index blocking factors and selectivities of all columns, relationships with respect to connections, join selectivities with respect to connections.
- Derived inputs: Coupling factors with respect to individual two-variable transactions. (These are derived from the data characteristics and the restriction predicates in the transactions.)

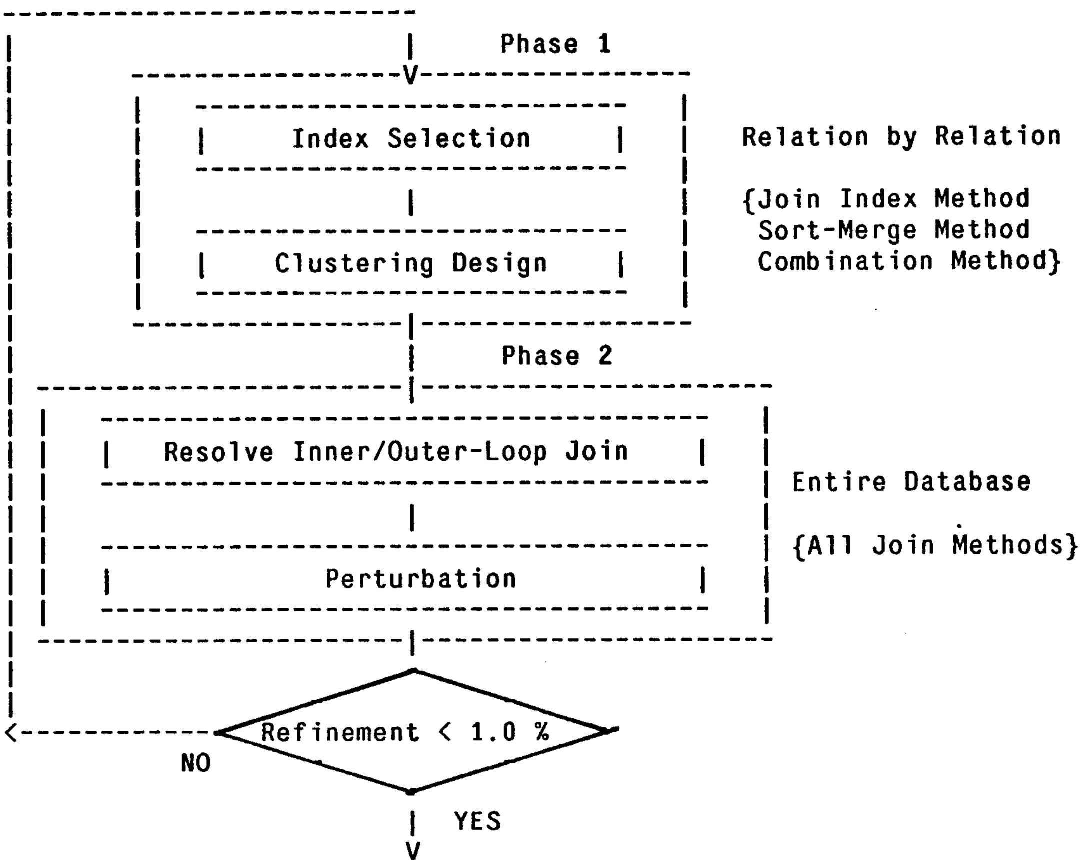


Figure 4-1: Algorithm 1 for the Optimal Design of Physical Databases.

Output:

- The optimal access configuration of the database, which consists of the optimal position of the clustering column and the optimal index set for each relation.

- The optimal join method for each two-variable transaction.

ALGORITHM 1

The design is performed in two phases: Phase 1 and Phase 2. These two phases are iterated until the refinement through the loop becomes negligible (say <1%). In Phase 1, based on the theory of separability, the access configuration is designed relation by relation independently of one another using only the join methods in the separable set—the join index method, the sort-merge method, and the combination method. Phase 1 is further divided into two steps: the Index Selection Step and the Clustering Design Step. In the Index Selection Step an optimal index set is chosen given the clustering column position determined in the Clustering Design Step of the last iteration. (Initially, in the first iteration, there is no clustering column.) In the Clustering Design Step, an optimal clustering column position is chosen given the index set determined in the Index Selection Step.

Before introducing the details for these steps, we define the function EVALCOST-1 as follows:

Function EVALCOST-1

Input:

- Access configuration of the relation being considered.
- Set of transactions that are to be processed in Phase 1 using the inner/outer-loop join method and the direction of the join for each transaction in the set. (These transactions are identified in Phase 2 of the previous iteration.)

Output:

- Total cost of the relation.

(In the input specification of this function as well as the functions or algorithms introduced later, the global input information introduced at the beginning of this section is implicitly assumed unless stated otherwise.)

The total cost of a relation is obtained by summing up the costs of single-relation transactions and the partial-join costs of two-relation transactions that refer to the relation. The cost of each transaction must be multiplied by its frequency. For each partial-join, the best partial-join algorithm is selected and its cost calculated. However, if the transaction is supposed to be processed by the inner/outer-loop join method according to the input information, that method will be used unconditionally according to the join direction specified because the inner/outer-loop join method cannot be treated uniformly with separable join methods in Phase 1 due to its nonseparable nature.

Using the function EVALCOST-1 defined above, the algorithm for index selection is described as follows:

Index Selection Step

Input:

- Clustering column position for each relation
- Set of transactions that are to be processed using the inner/outer-loop join method and the direction of the join for each transaction in the set.

Output:

- The optimal index set for each relation with respect to the input information.

Algorithm:

1. Pick one relation and start with an access configuration having a full index set.
2. Try to drop one index at a time and apply EVALCOST-1 to the resulting access configuration to find the index that yields the maximum cost benefit when dropped.
3. Drop that index.
4. Repeat Steps 2 and 3 until there is no further reduction in the cost.
5. Try to drop two indexes at a time and apply EVALCOST-1 to the resulting access configuration to find the index pair that yields the maximum cost benefit when dropped.
6. Drop that pair.
7. Repeat Steps 5 and 6 until there is no further reduction in the cost.
8. Repeat Steps 5, 6, and 7 with three indexes, four indexes, ..., up to k (k must be predefined) indexes at a time.
9. Repeat the entire procedure for every relation in the database.

The variable k, the maximum number of indexes that are dropped together at a time, must be supplied to the algorithm by the user. According to the results of the experiments, however, $k=2$ suffices in most practical cases.

The index selection algorithm presented here bears some resemblance to the one introduced by Hammer and Chan [HAM 76], but it uses the Drop Heuristic [FEL 66] instead of the ADD Heuristic [KUE 63]. The Drop Heuristic attempts to obtain an optimal solution by incrementally dropping indexes starting with a full index set. On the other hand, the ADD Heuristic adds indexes incrementally starting from an initial configuration without any index to reach an optimal solution. Since we are pursuing a heuristic approach (DROP heuristic) for index selection, the actual result is suboptimal. An experimental study in Appendix F shows that the algorithm finds optimal solutions in most of the cases.

The Clustering Design Step comes next in Phase 1.

Clustering Design Step

Input:

- Index set for each relation determined in the Index Selection Step.
- Set of transactions that are to be processed using the inner/outer-loop join method, and the directions of the join for each transaction in the set.

Output:

- Optimal position of the clustering column for each relation with respect to the input information.

Algorithm:

1. Select one relation.
2. Assign the clustering property to one column of the relation.
3. Apply EVALCOST-1 to the resulting access configuration.
4. Shift the clustering property to another column of the relation and repeat Steps 2 and 3.
5. Repeat Step 4 until all the columns of the relation have been considered, including the configuration having no clustering column is also considered. Then determine the one that gives the minimal cost as the clustering column (or none).

In Substep 2 the clustering property accompanies an index if the column has not been assigned one in the Index Selection Step. This strategy slightly enhances the accuracy of the design algorithms. More details on this strategy as well as other strategies enhancing the accuracy can be found in Appendix J.1.

The clustering design algorithm amounts to an enumeration of all possible alternatives. However, because of the restriction that a relation can have at most one clustering column, the time complexity is only linear on the number of columns in the relation. When a virtual column is involved, there could be more than one clustering column in a relation since the first component column of a virtual column that is clustering is itself a clustering column. But, since the two columns are tightly interlocked, the time complexity is still linear on the number of columns (now including virtual columns) in the relation.

In Phase 2 the design algorithm is extended to include the inner/outer-loop join method. Since the inner/outer-loop join method is nonseparable, it cannot be incorporated in Phase 1. Instead, a separate step (Resolve Inner/Outer-Loop Join Step) is attached to take a corrective action. Given the access configuration from Phase 1, for each two-relation transaction, the best join method is selected. If the inner/outer-loop join method happens to be the best one, it is remembered that the transaction be processed by the inner/outer-loop join method in Phase 1 of the next iteration. Also remembered is the direction of the join. To describe the algorithm for the Resolve Inner/Outer-Loop Join Step, we define the function EVALCOST-2.

Function EVALCOST-2

Input:

- Access configuration of the entire database.

Output:

- Total cost of the database.

Side Effect:

- Two-relation transactions that use the inner/outer-loop join method are marked, and their join directions recorded.

The total cost of the database is obtained by summing up the costs of all transactions multiplied by their respective frequencies. For each two-relation transaction, the best join method (including the inner/outer-loop join method) is selected and its cost calculated. As a side effect, if the best join method for a transaction is the inner/outer-loop join method, a reminder is attached to the transaction that it must be processed by the inner/outer-loop join method in Phase 1 of the next iteration. This reminder is one of the elements that interfaces Phase 1 and Phase 2 conveying information from one phase to another.

The following is the algorithm for Resolve Inner/Outer-Loop Join Step:

Resolve Inner/Outer-Loop Join Step

Input:

- The access configuration of the database produced by Phase 1.

Output:

- Set of transactions to be processed by the inner/outer-loop join method and the direction of the join for each transaction in the set.

Algorithm:

1. Apply EVALCOST-2 once. The desired output will be obtained by the side effects of EVALCOST-2.

The second step of Phase 2 is the Perturbation Step. This step eliminates snags in the design process which may be incurred by some anomalies.

One anomaly is due to the peculiar characteristics of update transactions; that is, in processing an update transaction, the join index always remains after Phase 1 during the first iteration because the join index method is the only one available to resolve the join predicate for the relation being updated. (The sort-merge method is not allowed for the relation to be updated; the inner/outer-loop join method cannot be used in Phase 1 of the first iteration.) A problem arises in the Resolve Inner/Outer-Loop Join Step when the inner/outer-loop join is costlier than the join index method, but less costly if the maintenance (update) cost of the join index is incorporated. In this situation it would be more beneficial to use the inner/outer-loop join method and drop the join index. But, since the Inner/Outer-Loop Join Step does not incorporate the index maintenance cost, the algorithm finds the join index method less costly and lets the join index stay. Hence, we may never have a chance to drop the index. Simply adding the maintenance cost to that of the join index method will not work since the maintenance cost of an index must be shared by all transactions accessing that index. Therefore, in the Perturbation step, we try to drop the join index and compare the total transaction processing costs before and after the change. If the change proves to be beneficial, the join index is actually dropped.

Another anomaly occurs because we consider the inner/outer-loop join method separately from the other join methods. Sometimes the presence of an index favors performing the inner/outer-loop join in a certain direction. Dropping that index and reversing the direction of the inner/outer-loop join, however, may be more beneficial. But, it is impossible to consider this alternative in the Inner/Outer-Loop Join Step since that step is not allowed to change the access configuration. To solve this problem, in the Perturbation Step, we also try to drop an arbitrary index (as well as join indexes) and make the change permanent if it reduces the cost.

We generalize this concept and try to *add* an index as well as to *drop* one. Here, the algorithm for the Perturbation Step of Algorithm 1 follows:

Perturbation Step

Input:

- Access configuration produced by Phase 1.
- Total cost of the database obtained in the Inner/Outer-Loop Join Step.

Output:

- Modified access configuration of the database.

Algorithm:

1. Pick a column in the database. Try to drop the index if the column has one; otherwise add one.
2. Obtain the total cost of the database using EVALCOST-2. If the change reduces the cost, make it permanent.
3. Repeat Steps 1 and 2 for every column in the database.

We note that the Perturbation Step is supposed to accomplish a minor revision in the current access configuration to eliminate the snags that obstruct a smooth flow of the design process. Thus, only a small number of columns will be affected by the Perturbation Step; the affected columns must be sparsely scattered, and relatively independent of one another. Accordingly, dropping or adding two or more indexes together is excluded from consideration. For the same reason, an arbitrary order is chosen in considering the columns.

4.3 Time Complexity of the Design Algorithm

The time complexity is estimated in terms of the number of calls to the cost evaluator (EVALCOST-1 or EVALCOST-2) which is the costliest operation in the design process. The overall time complexity of Algorithm 1 is $O(t \times v^{k+1}) + O(t \times c)$, where t is the number of transactions specified in the usage information, v the average number of columns in a relation, c the number of columns in the entire database, and k the maximum number of columns considered together in the Index Selection Step. Phase 1 contributes to the first term in the complexity; Phase 2 to the second.

Among the two design steps in Phase 1, the Clustering Design Step has a time complexity $O(t \times v)$ which is dominated by that of the Index Selection Step. In the Index Selection Step EVALCOST-1 is called for every k-combination of columns of the relation being considered and for every transaction that refers to the relation. This contributes the order of $(s/r) \times t \times v^k$, where r is the number of relations in the database and s is the average number of relations that a transaction refers to. (Thus, (s/r) represents the average ratio of the number of transactions referring to a particular relation to the total number of transactions.) This procedure is repeated until there is no further reduction in the cost (the number of repetitions is proportional to v). Since the entire procedure is repeated for every relation, the overall time complexity of Phase 1 is $O(t \times v^{k+1})$ if we assume that s is relatively fixed. More detailed derivation of the time complexity of the Index Selection Step can be found in Appendix D.

In Phase 2, the Resolve Inner/Outer-Loop Join Step requires only one call to EVALCOST-2; thus, it is dominated by the Perturbation Step. The Perturbation Step calls EVALCOST-2 for every column in the database and for every transaction in the usage. As a result, the time complexity of this step is $O(t \times c)$. Let us note that if v, the average number of columns in a relation, is relatively fixed, the time complexity of Algorithm 1 is linear on c, the total number of columns in the database.

Let us note that Algorithm 1 achieves a substantial improvement in time complexity compared with the exhaustive-search method whose time complexity is $O(t \times (v+1)^r \times 2^c)$. Here, the factor $(v+1)^r$ is the total number of clustering configurations since the clustering column could be any one of v columns of a relation or there could be no clustering column at all. The factor 2^c is the total number of index configurations since each of c columns in the database can either have an index or not.

4.4 Validation of Design Algorithms

An important task in developing a heuristic algorithm is its validation. Because physical database design is such a complex problem, finding mathematical worst-case bounds on the deviations from the optimality (we shall simply call them deviations) of the solutions produced by the heuristic algorithm is almost impossible. Consequently, we have to rely on empirical test results of the algorithm for its validation. A simple method would be to compare the heuristic solutions with the optimal ones for various input situations. In many cases, however, identifying the optimal solution itself is a difficult, often impossible, task. For simple situations optimal solutions can be obtained by exhaustively searching through all the possible alternatives. For more complex situations, however, an exhaustive-search is practically prohibited by its exponentially increasing complexity.

One alternative method for validating a heuristic algorithm in these complex cases is to devise different heuristic algorithms and compare their solutions. If these solutions are identical, we conclude that they are very likely to be optimal, for it is very unlikely that different heuristics can cause exactly the same deviations from the optimal solution. Thus, for this purpose, two additional design algorithms (see Figures 4-2 and 4-3) are proposed. The two algorithms are derived from Algorithm 1 introducing variations that help validate heuristics involved. We first introduce the algorithms and then compare them for the purpose of validation.

ALGORITHM 2

Algorithm 2 is almost identical to Algorithm 1 except that the two steps in Phase 1 are combined in one design step: the Combined Index Selection and Clustering Design Step (Combined Step). The algorithm is described below:

Combined Step

Input:

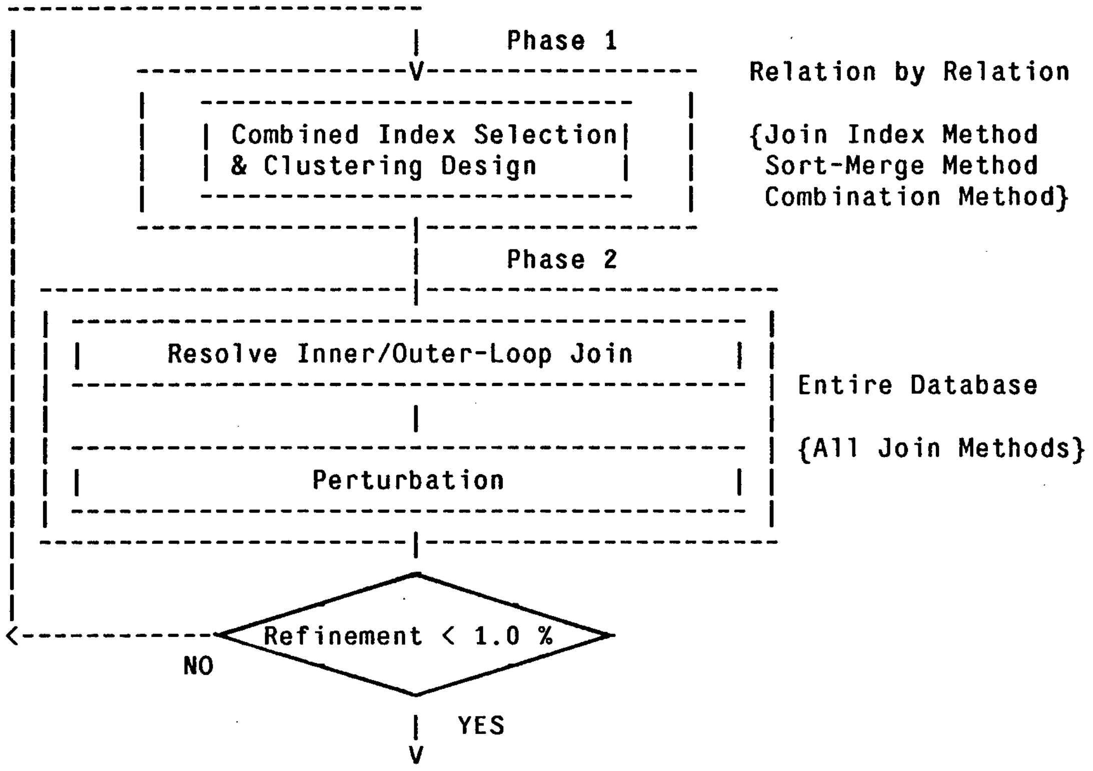

Phase 1
Relation by Relation
Combined Index Selection & Clustering Design
{Join Index Method Sort-Merge Method Combination Method}
Phase 2
Resolve Inner/Outer-Loop Join
Entire Database
{All Join Methods}
Perturbation
Refinement < 1.0 %
NO
YES

Figure 4-2: Algorithm 2 for the Optimal Design of Physical Databases.

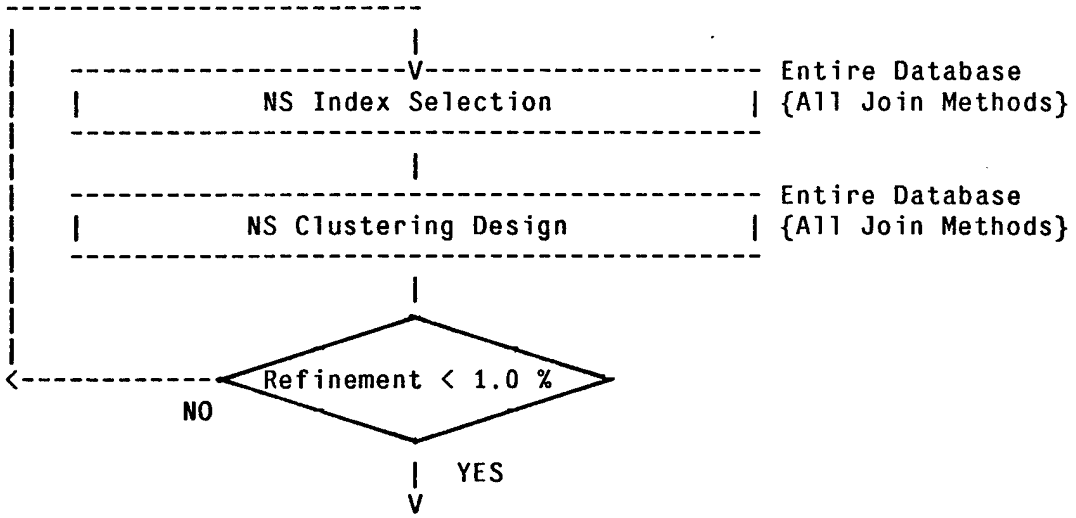


Figure 4-3: Algorithm 3 for the Optimal Design of Physical Databases.

- Set of transactions that are to be processed using the inner/outer-loop join method and the direction of the join for each transaction in the set.

Output:

- Optimal access configuration for each relation with respect to the input information.

Algorithm:

1. For each clustering column position in a relation, perform index selection as defined in Algorithm 1.
2. Save the best configuration.

ALGORITHM 3

Algorithm 3 is different from Algorithms 1 and 2 in that it does not rely on the property of separability. This algorithm has a much higher time complexity compared with the two previous algorithms (see Appendix D). The algorithm consists of one phase which, in turn, is decomposed into two steps: the NS Index Selection Step and the NS Clustering Design Step (the prefix NS stands for "nonseparable"). The two steps design the access configuration of the entire database all together rather than relation by relation. All available join methods are incorporated. The algorithms are described below:

NS Index Selection Step

Input:

- Clustering column positions determined in the NS Clustering Design Step of the last iteration.

Output:

- Optimal index set of entire database with respect to the given clustering column positions.

Algorithm:

1. Identical to the Index Selection Step except that the index set is designed for the entire database at the same time and using the function EVALCOST-2.

NS Clustering Design Step

Input:

- Index set of the database determined in the NS Index Selection Step.

Output:

- Optimal positions of the clustering columns with respect to the given index set.

Algorithm:

1. Start with an access configuration having no clustering columns.
2. Try to assign the clustering property to one column in the database at a time. Applying EVALCOST-2, find the column that yields the maximum cost benefit.
3. Assign the clustering property to that column.
4. Repeat Steps 2 and 3 with the constraint that one relation can have at most one clustering column until there is no further reduction in the cost.
5. Starting with the access configuration from Step 4, try to assign the clustering property to two columns in the database at a time. One relation can have at most one clustering column. Applying EVALCOST-2, find the pair that yields the maximum cost benefit.
6. Assign the clustering property to that pair.
7. Repeat Steps 5 and 6 until there is no reduction in the cost.
8. Repeat Steps 5, 6, and 7 with three columns, four columns, ..., up to k columns (k must be predefined) at a time.

The two algorithms are used for the validation of heuristics as follows. Algorithm 2 combines the two steps in Phase 1 into one design step. Thus, the heuristic of separating two steps in Algorithm 1 can be validated by comparing the solutions from Algorithms 1 and 2. Similarly, since Algorithm 3 does not exploit the property of separability, the inner/outer-loop join can be incorporated in Phase 1, and Phase 2 is no longer necessary. Thus, the heuristic involved to incorporate the inner/outer-loop join method in Algorithm 1 can be validated by comparing the solutions of Algorithms 1 and 3. Experimental studies for validation of the physical design algorithms can be found in Appendix D.

Another heuristic employed in all Algorithms is that of index selection. Since the DROP index selection heuristic is used in three algorithms in common, it cannot be validated by comparing these algorithms. Instead, since index selection is a relatively independent submodule in physical design algorithms, it can be validated separately from the other part of the design algorithms. With reasonably sized input situations, the exhaustive-search method is feasible to find the optimal solutions for this problem. Experimental studies for validation of index selection heuristic can be found in Appendix F.

4.5 Summary

An algorithm for the optimal design of multifile physical databases has been presented. This algorithm is based on the theory of separability and is heuristically extended to include the inner/outer-loop join method which is a nonseparable join method. Other nonseparable join methods, if available, can be incorporated similarly. The time complexity of this algorithm shows a significant improvement compared with that of the exhaustive-search method.

Two additional algorithms have been proposed for the validation of heuristics employed in the design algorithm. The validation can be performed by comparing the solutions of three algorithms that utilize different heuristics.

5. Index Selection

5.1 Introduction

We consider here and in Appendix F the problem of selecting a set of indexes that minimizes the transaction-processing cost in relational databases. Appendix F contains detailed experimental data and their analysis. Index selection is an interesting and well-defined subproblem of the access structure selection problem. For this reason, we isolate this problem from the rest of the access structure selection problem and concentrate on its own aspects.

Although there has been a considerable effort in the development of algorithms for index selection, most research in the past has concentrated on single-file cases. Furthermore, most of them addressed only secondary index selection, and incorporation of the primary structure (the clustering property) of the file has remained to be solved. In this chapter we develop an index selection algorithm with a reasonable efficiency that can be extended to multiple-file databases as well as extended to incorporate the clustering property.

We begin in Section 5.2 with the index selection algorithm for single-file databases without the clustering property. This algorithm is extended in Section 5.3 to incorporate the clustering property. An extension to the multiple-file environments is discussed in Section 5.4.

5.2 Index Selection for Single-File Databases

ALGORITHM 4

Input:

- Usage information: A set of various queries and update, insertion, deletion transactions with their relative frequencies.
- Data characteristics: Relation cardinality, blocking factor, selectivities and index blocking factors of all columns.

Output:

- The optimal (or suboptimal) index set.

Algorithm:

1. Start with a full index set.
2. Try to drop one index at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index that yields the maximum cost benefit when dropped.
3. Drop that index.
4. Repeat Steps 2 and 3 until there is no further reduction in the cost.
5. Try to drop two indexes at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index pair that yields the maximum cost benefit when dropped.
6. Drop that pair.
7. Repeat Steps 5 and 6 until there is no further reduction in the cost.
8. Repeat Steps 5, 6, and 7 with three indexes, four indexes, ..., up to k (k must be predefined) indexes at a time.

The variable k, the maximum number of indexes that are dropped together at a time, must be supplied to the algorithm by the user. According to the results of the experiments, however, $k=2$ suffices in most practical cases.

The algorithm presented bears some resemblance to the one introduced by Hammer and Chan [HAM 76], but with one major modification: the DROP heuristic [FEL 66] is employed instead of the ADD heuristic [KUE 63]. The DROP heuristic attempts to obtain an optimal solution by incrementally dropping indexes starting from a full index set. On the other hand, the ADD heuristic adds indexes incrementally starting from an initial configuration without any index to reach an optimal solution. An experimental study in Appendix F shows that the solutions generated by the DROP heuristic are close to the optimal in many practical situations. It also indicates that the

DROP heuristic performs better than ADD heuristic. The following argument provides one possible reason for this result. In the ADD heuristic, when the first index is added, the cost changes drastically causing an abrupt change in the design process. In the DROP heuristic, however, dropping indexes causes a smooth transition in the design process since dropping one index does not make a big change in the cost due to the compensating effect of the other existing indexes. Advantages of the DROP heuristic over the ADD heuristic in the warehouse location problem are summarized in [FEL 66].

The time complexity of the algorithm is $O(g \times v^{k+1})$, where g is the number of transactions specified in the usage information, v the number of columns in the relation, and k the maximum number of columns considered together in the algorithm. The time complexity is estimated in terms of the number of calls to the cost evaluator which is the costliest operation in the design process. In the algorithm the cost evaluator is called for every k-combination of columns of the relation, and for every transaction in the usage information. This contributes the order of $g \times v^k$. The procedure is repeated until there is no further reduction in the cost. Since the number of repetitions is proportional to v, the overall time complexity is $O(g \times v^{k+1})$.

5.3 Index Selection when the Clustering Column Exists

Incorporation of the clustering property to the index selection algorithm is straightforward. Two algorithms for this extension are presented below:

ALGORITHM 5

1. For each possible clustering column in the relation perform index selection.
2. Save the best configuration.

ALGORITHM 6

1. Steps 2 and 3 are iterated until the improvement in the cost through the iteration loop is less than a predefined value (e.g., 1%).

2. Perform index selection with the clustering column determined in Step 2 of the last iteration. (During the first iteration it is assumed that there is no clustering column.)

3. Perform clustering design with the index set determined in Step 1. The clustering property is assigned to each column in turn, and the best clustering column is selected.

Algorithm 5 is a pseudo enumeration since index selection is repeated for every possible clustering column position. Accordingly, Algorithm 5 has a higher time complexity compared to Algorithm 6, but has a better chance of finding an optimal solution. Algorithm 5 corresponds to Phase 1 of Algorithm 2, a physical database design algorithm presented in Chapter 4. Algorithm 6 corresponds to Phase 1 of Algorithm 1.

5.4 Index Selection for Multiple-File Databases

Extension of the index selection algorithm for application to multiple-file databases is also straightforward. The extended algorithm (let us call it Algorithm 7) is identical to Algorithm 4 except for the following considerations:

1. In all steps the entire database is designed at the same time. It is done by treating all columns in the database uniformly as if they were in a single relation.

2. In Steps 2 and 5, when evaluating transactions involving more than one relation, the optimizer [SEL 79], [STO 76] has to be invoked to find the optimal sequence of access operations.

Algorithm 7, if the clustering property is incorporated, corresponds to Algorithm 3 presented in Chapter 4.

5.5 Summary

In this chapter the access structure selection problem has been analyzed from the view point of the index selection problem. Important components of physical database design algorithms – Phase 1 of Algorithm 1, Phase 1 of Algorithm 2, and Algorithm 3 itself – have been shown to be extensions of index selection algorithm. The advantages of the DROP heuristic over the ADD heuristic have been discussed.

6. Transaction-Processing Costs in Relational Database Systems

6.1 Summary

This chapter is identical to Appendix E. We therefore present here only a brief summary of the chapter. Accurate estimation of transaction costs is important for both query optimization and physical database design. In this chapter a comprehensive set of formulas for estimating transaction-processing costs in relational database systems is developed. The assumptions and the model of storage structures considered are stated in detail in Appendix E.2. The experiments for the design algorithms introduced in Chapter 4 have been performed using the cost formulas developed in this chapter. However, let us note that the theory presented in Chapter 2 and Chapter 3 do not depend on the specific cost model.

In this chapter, first a set of necessary terminology is defined to provide a mechanism for understanding interaction among relations in multiple-file environments. Next, a set of elementary cost formulas is developed for elementary access operations. In doing so, four types of ordering are defined to characterize the order of accessing tuples. Finally, transactions are classified into eight types, and the cost formulas for each type are derived as composites of elementary cost formulas. The cost formulas have been fully implemented in the Physical Database Design Optimizer introduced in Appendix D. The detailed discussions for developing cost formulas are referred to Appendix E.

7. Estimating Block Accesses in Database Organizations

7.1 Summary

This chapter is identical to Appendix C. Thus, we present here only a brief summary of the chapter.

An approximation formula is developed for estimating the number of block accesses when randomly selected tuples are accessed in TID order. This formula improves Yao's exact formula in the sense that it significantly reduces the computation time by eliminating the iterative loop, while providing a practically negligible deviation (maximum error$=$3.7%) from the exact formula. It also significantly improves Cardenas' earlier formula, which has a maximum deviation of $e^{-1}=36.8\%$.

The formula is presented below without derivation. The details of the development of this formula are referred to Appendix C.

Block access formula: Let n records be grouped into m blocks ($1 \leq m \leq n$), each containing $p=n/m$ records. If k records are randomly selected from the n records, the expected number of blocks hit (blocks with at least one record selected) is given by

$$\begin{aligned} b_{w1}(m,p,k)/m = {} & [1 - (1-1/m)^{k}] \\ & + [1/m^{2}p \times k(k-1)/2 \times (1-1/m)^{k-1}] \\ & + [1.5/m^{3}p^{4} \times k(k-1)(2k-1)/6 \times (1-1/m)^{k-1}] \\ & \qquad \text{when } k \leq n-p, \text{ and} \\ b_{w1}(m,p,k)/m = {} & 1 \qquad \text{when } k > n-p \end{aligned}$$

8. Separability in Network Model Database Systems

8.1 Summary

This chapter is identical to Appendix B. Thus, we present here only a brief summary of the chapter. We discuss an application of the theory of separability to network model database systems. In particular, we show that a large subset of practically important access structures provided by the network model database systems has the property of separability under the usage specification scheme proposed. The implication is that, if the available access structures are restricted to this subset, the optimal design of the access configuration of a multifile record type database can be reduced to the collective optimal designs of individual record types. The physical designs thus obtained is then extended, using heuristics, to include other access structures that have not been incorporated initially. The CODASYL '78 Database Specification [COD-a 78] [COD-b 78] is used as the environment for our discussion. The major assumptions and detailed discussion on this subject are referred to Appendix B.

9. Design Algorithms for More-than-Two-Variable Transactions

So far only transactions that involve at most two variables have been considered. Transactions of more than two variables can be incorporated in the physical design methodology through decomposition into a sequence of two-variable transactions. In this chapter a preliminary —though not comprehensive—methodology is suggested for multivariable transactions. We investigate some potential problems that violate the conditions for separability and discuss approximations to solve those violations. However, a complete treatment of this problem including the validation of heuristics involved needs much more work to be done and is left as a further study. In Section 9.1 an extended algorithm for relational databases is discussed; in Section 9.2 one for network model databases is discussed.

9.1 An Extended Algorithm for Relational Databases

9.1.1 The Algorithm

ALGORITHM 8

1. Start with an initial access configuration in which every column has an index and the clustering property.

2. Decompose multivariable transactions into optimal sequences of two-variable transactions based on the current access configuration.

3. Invoke the physical database design algorithm for two-variable transactions.

4. Repeat Steps 2 and 3 until the variation in the total cost becomes smaller than a predefined value (say 1%).

The algorithm starts with an initial configuration in which every column has an index and the clustering property. The initial configuration is intended to be as close to the optimal solution as possible. In particular, it is believed that this configuration is closer to the optimal than the one having a full index set but without any clustering property. Let us note that this initial configuration is not a practically feasible one; but, once the first iteration is finished, the access configuration becomes feasible.

9.1.2 Decomposition

Decomposition is a procedure that finds an optimal sequence of two-variable transactions, or equivalently, an optimal join sequence. In principle an optimal join sequence can be obtained by enumerating all permutations of relations to be joined. Since the number of permutations could be extensive, a heuristic is used to restrict the search space [SEL 79]. When possible, the search is reduced by considering only those sequences in which a relation is related by a join predicate to any of previous relations in the sequence. More formally speaking, in joining relations R_1, R_2,...,R_n, only those sequences R_{i1}, R_{i2},...,R_{in} are examined which, for all j (j=2,...,n), satisfy either of the following conditions:

1. R_{ij} has at least one join predicate with some relation R_{ik}, where $k < j$.
2. for all $k > j$, R_{ik} has no join predicate with any of R_{i1}, R_{i2}, ...,$R_{i(j-1)}$.

The intention of this heuristic is to defer all joins requiring cartesian products as long as possible. More discussions on this heuristic can be found in [SEL 79].

A join sequence can be visualized as a sequence of two-variable transactions as follows. Suppose we have a join sequence R_1, R_2,...,R_n. Then, the corresponding sequence of two-variable transactions is (R_1 JOIN R_2), (T_2 JOIN R_3), (T_3 JOIN R_4),...,(T_{n-1} JOIN R_n), where T_i is the result of R_1 JOIN R_2 JOIN ... JOIN R_{i-1}. Thus, except for the first join, each two-variable transaction is a join between a temporary relation that contains the result of the joins performed so far and the next relation in the join sequence.

A temporary relation can be either materialized or nonmaterialized. When materialized, a temporary relation is written in a file on the secondary storage. When not materialized, a temporary relation is a relation only in concept and does not physically exists. For instance, if R_1, R_2, R_3 are joined by using the inner/outer-loop join method recursively (i.e., for one tuple of R_1 corresponding

R_2 tuples, and in turn, corresponding R_3 tuples are retrieved; this procedure is repeated for every tuple of R_1), the temporary relation R_1 JOIN R_2 is not materialized, but still we can conceptually visualize the temporary relation T_2 as the result that would be obtained by joining R_1 and R_2 only.

Materialized or not, a temporary relation has its cardinality which we call the *result cardinality*. The result cardinality up to relation R_j can be estimated as follows:

Result cardinality =

$\Pi_{i=1}^{j}$ ($n_i \times$selectivity of the restriction predicate for R_i)$\times$

$\Pi_{\text{for each join}}$(1/column cardinality of the join column of the 1-side relation),

where n_i is the cardinality of relation R_i. This assumes that each distinct join column value in the N-side relation of the 1-to-N relationship has a matching value in the join column of the 1-side relation according to the rules in the structural model [WIE 79].

For decomposed two-variable transactions involving temporary relations, only the following join methods can be used. Let R_1 be the temporary relation.

R_1	R_2
1. Sort-merge method (partial)	Sort-merge method (partial)
2. Sort-merge method (partial)	Join index method (partial)
3. Inner/Outer-Loop Join Method(partial) (from R_1)	Inner/Outer-Loop Join Method(partial) (to R_2)

The join index method (partial) for R_1 is excluded from consideration since a temporary relation does not have any index unless one is explicitly created. Since creating an index at run time is an expensive procedure, we exclude this possibility. For the same reason, the Inner/Outer-Loop join method is prohibited from R_2 to R_1.

The partial-join costs of these join methods for decomposed two-variable transactions are slightly different from the ordinary ones. For the first two combinations the temporary relation must be materialized. Therefore, the partial-join cost of R_1 must include the cost of writing the temporary relation to the disk initially. On the other hand, when the third combination is used, the temporary relation need not be materialized,-and further, it need not be read in since necessary tuples are already held in the main memory. Thus, the partial join cost of R_1 becomes 0.

For convenience, we make further modification to the definition of partial-join costs for decomposed two-variable transactions. Since we are not concerned about designing access structures for R_1, and further the partial-join algorithm for R_1 is totally dependent on the partial-join algorithm for R_2, we can safely combine the partial-join cost of R_1 with that of R_2. This way, we do not have to consider the cost of the temporary relation separately. Thus, the modified partial-join cost for R_2 can be calculated as follows:

Modified Cost of the Sort-Merge Method (partial) for R_2
= Cost of the Sort-Merge Method (partial) for R_2
+ Cost of materializing R_1
+ Cost of the Sort-Merge Method (partial) for R_1

Modified Cost of the Join Index Method (partial) for R_2
= Cost of the Join Index Method (partial) for R_2
+ Cost of materializing R_1
+ Cost of the Sort-Merge Method (partial) for R_1

Modified Cost of the Inner/Outer-Loop Join Method (partial) for R_2
= Cost of Inner/Outer-Loop Join Method (partial) for R_2

9.1.3 Discussion

In this subsection we shall investigate a potential problem in decomposition that violates a condition for separability in decomposing a multivariable join in relational database systems. First we identify the problem and propose a simple solution. It turns out that the simplest solution is to ignore the problem. We shall provide some justification (though not complete) for this approach.

So far, we modelled a decomposed two-variable join as a join between a temporary relation – materialized or not – representing the result of the joins already performed and the next relation in the join sequence. If the inner/outer-loop join method is used, however, there are some cases in which the cost calculated based on this model is different from the actual cost as we see in Example 9.1.

Example 9.1: Let R_1, R_2, R_3, and R_4 be four relations having N-to-1 relationship as described in Figure 9-1.

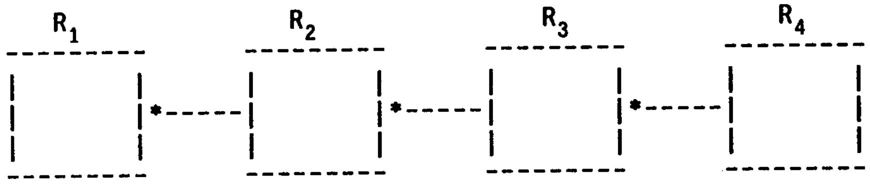


Figure 9-1: Four Relations. The symbol *— — stands for an N-to-1 relationship.

We consider the join sequence $\langle R_1, R_2, R_3, R_4 \rangle$. Suppose that the join column of R_1 is clustered. If the inner/outer-loop join method is used for R_1, the tuples of R_1 having the same join column value (let us call them a group) that satisfy the restriction predicate for R_1 will be accessed consecutively. Accordingly, the same tuple in R_2 having the same join column value will be repeatedly accessed; thus, the block containing this tuple will very likely reside in the main memory without incurring additional I/O accesses. Furthermore, the tuple in R_3 matching the R_2 tuple and accordingly the R_4 tuple matching the R_3 tuple will also be repeatedly accessed causing the blocks containing these tuples to remain in the buffer. Thus, effectively, the cost of the inner/outer-loop join method for R_3 is reduced by a factor equivalent to the average number of tuples of R_1 in the same group that satisfy the restriction predicate for R_1. The same situation happens when the join index method or the sort-merge method is used for R_1. It also happens to R_3 and R_4 when temporary relation T_2 (R_1 JOIN R_2) is materialized and the sort-merge method is used for T_2. □

The situation in Example 9.1 violates a condition for separability. When a multivariable join is decomposed, the access configuration of, or the join methods to be used for, the previous relations in the sequence are not known. Therefore, there is no way to find out whether the cost of the inner/outer-loop join for a decomposed two-variable join would be reduced to allow for repeated accesses.

As a simple solution to this problem, we keep the temporary-relation view for the nonmaterialized intermediate result. By doing that, we sometimes overestimates the cost of the inner/outer-loop join method; but, the property of separability is preserved. However, we believe the error that might be introduced by this approximation is not significant according to the following justification. To illustrate, let us again consider two relations R_1 and R_2 having an N-to-1 relationship. Let F_1 and F_2 be the selectivities of the restriction predicates for R_1 and R_2.

1. If $F_1 \times n_1 < n_2$ there are less tuples selected than the number of groups in R_1 assuming that there are not many dangling tuples in R_2; thus, most groups will have at most one selected tuple, and the repeated access problem will rarely occur.

2. If $F_1 \times n_1 > n_2$, in many cases performing the inner/outer-loop join from R_2 to R_1 is more beneficial because it reduces the number of traversals of SET occurrences. If this is the case, the repeated access problem will not occur since join is performed from 1-side to N-side relation of the 1-to-N relationship.

3. Sometimes, the inner/outer-loop join method cannot be performed from R_2 to R_1 (for instance, if R_1 is a temporary relation). For these cases justification 2 is not valid; instead, we make the following arguments: if $F_1 \times n_1 \geq n_2$, the cost of the join index method or the sort-merge method is comparable to or even less than the inner/outer-loop join cost for the following two reasons; thus, overestimating the cost of the inner/outer-loop join method by ignoring the repeated access problem will not affect the total transaction cost since we have less costly alternatives that will be chosen by the optimizer.

   a. Since $F_1 \times n_1 \geq n_2$, the number of tuples selected in R_1 is greater than or equal to the number of join column values, which is equal to the number of groups. Thus, most of the groups will be selected. Accordingly, most of the tuples as well as join index entries of R_2 will be accessed—possibly repeatedly. Hence, the cost of the join index method may be comparable or even less than that of the inner/outer-loop join method since, in the join index method, data tuples or index entries are accessed only once.

   b. Similarly, since a majority of R_2 tuples (or at least their index entries if tuples do not satisfy the restriction predicate) are accessed, at least one block access will be needed for every tuple in R_2. In this case the cost of the sort-merge method may be less than that of the inner/outer-loop join method.

These arguments show that the solution of simply ignoring the repeated access problem will not cause much deviation from the optimal in the transaction-procession cost.

9.2 An Extended Algorithm for Network Model Databases

9.2.1 Usage transformation functions

In this section we describe an algorithm to extend the physical design of network model databases to more-than-two-variable transactions. Specifically, we present a method for obtaining the usage transformation functions F_{OM} and F_{MO} defined in Chapter 8. These two functions together with function f_{ENT} represent the entire usage information to be used for the physical database design. The usage transformation functions were defined to transform the number of traversals of SET types, f_{OM} or f_{MO}, to the number of traversals of SET occurrences. For the purpose of this section, however, we define the usage transformation functions to transform the number of database entries $f_{ENT}(T,R)$ of transaction T to the number of traversals of SET occurrences. The two definitions of usage transformation functions are not inconsistent because f_{OM} and f_{MO} can be derived from f_{ENT} and the *access path tree* which will be defined shortly. We also eliminated the parameter PRED assuming that in a transaction only one database entry occurs.

To derive these functions, we introduce the concept of *access path tree* developed by Gerritsen [GER 77]. An access path tree represents the record types, connected by access paths, as well as the order of visiting them. It is derived consistent with the conceptual schema and is organized in such a way that the preorder traversal matches the order of visiting the nodes. Figure 9-2 shows an example of such a tree. The nodes marked R_1, R_2, etc. represents the record types. Access paths S_1, S_2 etc. correspond to the SET types. Associated with each record type R are its cardinality, n_R, and a predicate, $PRED_R$, that will be applied to its records. The point of entering the database is marked with DBENTER. In the access path tree we denote the SET type to which the subtree rooted on R_i (or R) is attached as S_i (or set(R)). The symbol '*' represents the member record type of a SET type.

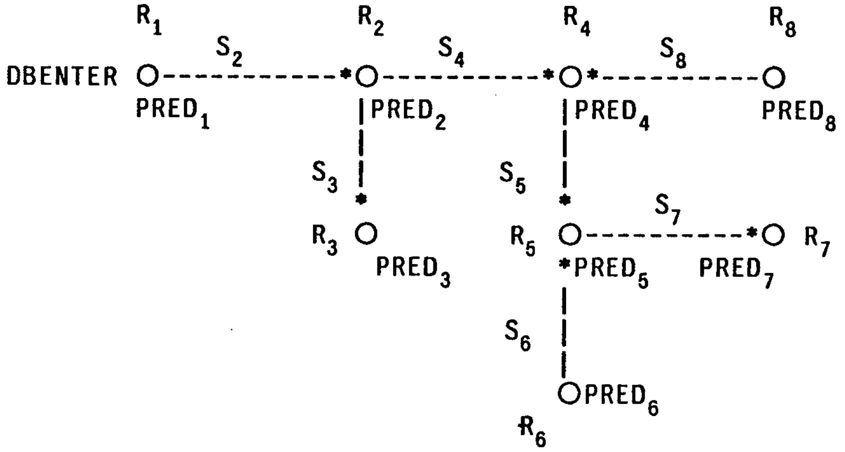


Figure 9-2: An Access Path Tree.

To achieve usage transformation we use the concept of *active records* of a SET type in the access path tree. The set of active records of a SET type corresponds to the result of processing the transaction had the record types in the subtree connected by the SET not existed in the access path tree. Accordingly, the number of active records determines the number of traversals of the SET occurrences to the next record type in the preorder traversal. Thus, the number of traversals of the SET occurrences is derived as follows:

$$F_{OM}(T,R,S) = f_{ENT}(T,R_1) \times \text{ACTIVE}(\text{set}(\text{owner}(R,S))) \qquad (9.1)$$

$$F_{MO}(T,R,S) = f_{ENT}(T,R_1) \times \text{ACTIVE}(\text{set}(R)) \qquad (9.2)$$

where ACTIVE(S) represents the number of active records of SET type S, and owner(R,S) the owner record type of R with respect to SET type S.

9.2.2 Number of active records

We now proceed to develop an algorithm to obtain the number of active records. We begin with a simple case and extend it to more complex cases. First, we assume that the access path tree is a linear list without any branch; then the number of active records of SET type S_{n+1} can be obtained as follows:

$$\text{ACTIVE}(S_{n+1}) = \text{ACTIVE}(S_n) \times J_{R_{n-1},S_n} \times g_{R_{n-1},S_n} \times \text{SEL}(\text{PRED}_n)$$

$\text{ACTIVE}(S_2) \quad = n_1 \times \text{SEL}(\text{PRED}_1)$

Example 9.2 further illustrates this case.

Example 9.2: Consider an access path tree in Figure 9-3.

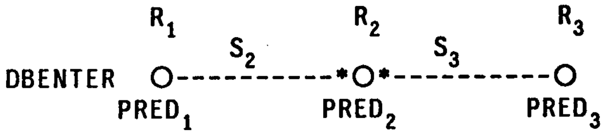


Figure 9-3: A Simple Access Path Tree without Any Branch.

Associated with the tree are the following data.

Cardinality	Grouping Factor	Join Selectivity	Selectivity of Predicate
$n_1=50$	$g_{R_1,S_2}=1$	$J_{R_1,S_2}=1$	$\text{SEL}(\text{PRED}_1)=0.1$
$n_2=200$	$g_{R_2,S_2}=4$	$J_{R_2,S_2}=1$	$\text{SEL}(\text{PRED}_2)=0.5$
$n_2=40$	$g_{R_2,S_3}=10$	$J_{R_2,S_3}=1$	$\text{SEL}(\text{PRED}_3)=0.5$
	$g_{R_3,S_3}=1$	$J_{R_3,S_3}=0.5$	

Then, the number of active records in R_1, R_2, and R_3 are

$\text{ACTIVE}(S_2) = 50 \times 0.1 = 5$

$\text{ACTIVE}(S_3) = 5 \times 1 \times 4 \times 0.5 = 10$ □

To extend the method of obtaining the number of active records to a more general access path tree (the tree is no longer is a linear list), we define procedure LABEL that traverses the tree in preorder, calculates the number of active records, and records the number in the global array of variables ACTIVE[S]. Here, function Root returns the root node of branch B_i. A call to LABEL(R_1,R_0) calculates the number of active records and sets the global variables ACTIVE[S] for all SET types. Here, two arrays of global variables, ACTIVE and TACTIVE, are used. TACTIVE[R] represents the number of active records when the tree traversal has been completed up to record type R; but, it changes every time the traversal of a branch of record type R is completed. ACTIVE[S] keeps the value of TACTIVE[R] just before SET type S leading to a branch of R is traversed. The parameter R-PREV represents the record type connected to R via SET set(R). Let us note that R-PREV is not the record type last visited.

To set up the initial conditions we create a hypothetical record type R_0 and SET type S_1 such that TACTIVE[R_0]=1, $J_{R_0,S_1}=1$, $J_{R_1,S_1}=1$, $g_{R_0,S_1}=1$, $g_{R_1,S_1}=n_1$. Equivalently, record type R_0 has one record that is linked to all the records of R_1 via SET type S_1.

```
procedure LABEL(R, R-PREV)

        begin
        TACTIVE[R]=TACTIVE[R-PREV]×SEL(PRED_R)×g_{R,S(R)}×J_{R-PREV,S(R)}
        for every branch B_i
                begin
                ACTIVE[set(Root(B_i))]=TACTIVE[R]
                LABEL(Root(B_i),R)
                TACTIVE[R]=TACTIVE[Root(B_i)]
                end
        end
```

9.2.3 Predicate branch

Procedure LABEL assumes that each record type in the access path tree contributes some data fields in the output. Sometimes, a branch in the tree is traversed only to check the existence of related records satisfying the specified predicates. We call this a *predicate* branch: it serves in its entirety as one predicate.

In this section we extend the procedure LABEL to incorporate predicate branches. The selectivity of a predicate branch is given by Ratio(Root(Branch)). To present the function Ratio, we first define function f that calculates the fraction of records of Father(R) to be selected when R has a restriction predicate having selectivity 'factor'. Function Father returns the father node of R in the access path tree.

```
function f(factor, R)
        begin
        if R has an 1-to-N relationship with its father then
                $f = factor \times J_{FATHER(R),S(R)}$

        if R has an N-to-1 relationship with its father then
                $f = (b(n_R/g_{R,S(R)}, g_{R,S(R)}, n_R \times factor) / (n_R/g_{R,S(R)})) \times J_{FATHER(R),S(R)}$
        end
```

In function f, if R has a 1-to-N relationship with its father, 'factor' and the linkage factor of father(R) is multiplied to obtain the fraction of records of Father(R) to be selected. On the other hand, if R has an N-to-1 relationship with its father, the number of set occurrences in R selected by 'factor' is obtained by using the 'b' function first, and the result is divided by the total number of SET occurrences in R to find the fraction of SET occurrences selected by the predicate; this fraction is multiplied by the linkage factor of father(R).

With this definition of function f, function Ratio is defined as follows:

```
function Ratio(R)
        if R is a leaf-node then
                $Ratio = SEL(PRED_R)$
        else
                $Ratio = SEL(PRED_R)$
                    $\times \prod_{\text{for each branch } B_i \text{ of } R} f(Ratio(Root(B_i)), Root(B_i))$
```

Function Ratio calculates the fraction of records of R to be selected according to all the predicates specified for the nodes in its subtree, as well as the predicate for R itself. If R is a leaf node, it has only its own predicate; thus, the value of the function is the selectivity of this predicate. If R is a nonleaf node, the effective selectivity of all its branches must be multiplied to $SEL(PRED_R)$.

Using functions f and Ratio, procedure LABEL is extended to handle the general case having predicate branches as follows:

```
procedure LABEL(R, R-PREV, flag)

        begin
        $TACTIVE[R] = TACTIVE[R\text{-}PREV] \times SEL(PRED_R) \times g_{R,S(R)} \times J_{R\text{-}PREV,S(R)}$
        flag = true
        for every branch $B_i$
            begin
            $ACTIVE[set(Root(B_i))] = TACTIVE[R]$
            $LABEL(Root(B_i),R,flag)$
            if flag then ($B_i$ is a predicate branch)
                $TACTIVE[R] = TACTIVE[R] \times f(Ratio(Root(B_i)))$
            else
                $TACTIVE[R] = TACTIVE[Root(B_i)]$
            end
        if any data item of R propagates to the result then flag = false
        end
```

In this procedure a reference parameter 'flag' indicates whether any data items propagate to the result from the branch B_i. If none does, then the branch is a predicate branch, and the current number of active records are reset to $TACTIVE[R] \times f(Ratio(Root(B_i)))$. TACTIVE[R] was the current number of active records just before the traversal of branch B_i started. $f(Ratio(Root(B_i)))$ is the effective selectivity for R of all the predicates in branch B_i. This procedure LABEL can handle the most general structure of the access path tree including predicate branches.

9.2.4 Discussion

In this subsection we shall investigate a potential problem that violates a condition for separability in extending the design algorithm to more-than-two-variable transactions for network model database systems. Just as in relational systems, it seems that the simplest solution is to ignore the problem. We shall provide some justification for this approximation solution.

The number of active records determines the number of traversals of SET occurrences. If traversals of the SET occurrences are totally random, we can consider them as independent traversals. However, in some cases the same SET occurrence is traversed more than once consecutively, and the repeated traversal cannot be considered independent. Specifically, we have this situation when the root node (R_1) of the access path tree is a member of a SET type (S), and the records of R_1 are accessed according to the order of values of linking data item of this SET type. This situation happens in the following cases.

1. The records of R_1 are clustered via set S. These records are accessed by an area scan.
2. The records of R_1 are accessed through a record order key defined on the linking data item.
3. The records of R_1 are associatively accessed through a key defined (that in turn can be implemented with an index for example) on the linking data item.

In this situation the records of R_1 in the same SET occurrence (let us call them a *group*) that satisfy the restriction predicate for R_1 are accessed consecutively. Accordingly, the corresponding owner record of type R_2 is repeatedly accessed, and the block containing this record will very likely reside in the main memory without incurring additional I/O accesses. Furthermore, the record of the next record type, R_3, matching R_2 record will also be repeatedly accessed, and the corresponding block will remain in buffer. Similarly, the records of record types in the rest of the access path tree that are directly or indirectly related to the records of R_1 will be repeatedly accessed reducing the number of I/O accesses. Let us note that we encountered a similar situation in relational systems when the inner/outer-loop join method was used.

This situation poses a problem when we design the access configurations of individual record types separately. In particular, when we design the access configuration of a record type (say R_3) other than R_1, there is no way of knowing which access structures R_1 would have or which access structures of R_1 will be exploited in processing a transaction. As a result, we cannot determine whether the number of I/O accesses will be reduced due to repeated accesses to the same records.

Although there is no clear solution for this problem, we have reasonable justifications for designing individual record types separately by simply ignoring the problem. Specifically, we believe that the error incurred by this approximation is not significant because it does not appear that the three exceptional cases stated above happen frequently for the following reasons:

1. An area scan is not and must not be used frequently. Thus, Case 1 in page 71 will not occur frequently.

2. Accessing the records of R_1 through a record order key requires scanning every record in R_1 regardless of the predicate specified for it. In this case, it frequently would be less costly to access owner records first and then access records of member records (R_1) through the SET because the predicate on R_2 can reduce the number of accesses to R_1. This reduces the possibility that Case 2 can happen.

3. The restriction predicate on linking data item frequently is specified for the owner record type; the reverse seems to be rare. For instance, suppose we have two record types EMPLOYEES and CHILDREN. The CHILDREN is the member record type, and the data item EMPLOYEE-NAME is the linking data item. Consider a query "Show AGE, JOB, DEPARTMENT of employee 'John Smith' and all his/her children." In this case it would be somewhat awkward to specify the predicate CHILDREN.EMPLOYEE-NAME = 'John Smith' rather than EMPLOYEES.EMPLOYEE-NAME = 'John Smith'. This reduces the possibility that Case 3 can happen.

4. Exceptional cases more rarely occur especially when the system is implemented according to the 1971 DBTG Proposal [COD 71]. In this proposal the record order key and the indexes do not exists. Thus, Cases 2 and 3 never arise, and exceptions can only occur when an area scan is used, R_1 is the clustered via SET which is to be traversed subsequently, and further R_1 is the member type of the SET. It is not likely that this situation occurs frequently.

10. Summary of the Research

10.1 Summary

A new approach to multifile physical database design was presented. Most previous approaches towards multifile physical database design concentrated on developing a cost evaluator and its application in the design aid systems. To accomplish the optimal physical design, however, this approach had to rely on the designer's intuition or, in the worst case, on an exhaustive search which is practically infeasible even for moderate-sized databases.

In our approach a theory was developed to partition the entire database design into collective subproblems. Straightforward heuristics were subsequently employed to incorporate features that could not be included in the theory. This approach is somewhat formal, deliberately avoiding excessive reliance on heuristics. Our purpose is to render the whole design phase manageable and to facilitate understanding of the underlying mechanisms.

In Chapter 2 we introduced the theory of separability. The theory identified the condition for separability under which the problem of optimal assignment of access structures to the entire database can be reduced to the subproblems of optimizing individual logical objects independently of one another.

Application of the theory to the relational database systems was discussed in Chapter 3. Specifically, it was shown that the set of join methods that consists of the join index method, the sort-merge method, and the combination of the two satisfies the conditions for separability under certain constraints. Thus, if the DBMS provides only these join methods, the physical database design can be partitioned into the designs of individual relations.

Application of the theory to the network model database systems was discussed in Chapter 8. As in relational systems, it was shown that a large subset of practically important access structures that are available in the network model database systems satisfies the conditions for separability.

In Chapter 4 three algorithms for the physical design of relational databases were proposed. Based on the concept of separability, Algorithm 1 and Algorithm 2 design the access configuration relation by relation. These algorithms were also extended using heuristics to incorporate the inner/outer-loop join method (a nonseparable join method). On the other hand, Algorithm 3 designs the configuration of the entire database all together. These algorithms were fully implemented in the Physical Database Design Optimizer (PhyDDO) and tested with simple situations. The result showed that all three algorithms found optimal solutions in most cases. Specifically, among the 21 input situations tested, Algorithm 1 found optimal solutions in 19 cases, Algorithm 2 in 21 cases which are all cases, and Algorithm 3 in 20 cases. Even in the cases in which nonoptimal solutions were found, the deviations were far from significant (maximum error = 6.6%).

Index selection algorithms for relational databases were presented in Chapter 5. An algorithm based on the DROP heuristic was introduced for single-file databases and compared with the ADD heuristic. In an exhaustive test performed, the DROP heuristic found optimal solutions in all cases. In comparison, the ADD heuristic found nonoptimal solutions in several occasions. This algorithm based on the Drop heuristic was extended to incorporate the clustering property and also extended for application to multifile databases.

A comprehensive set of cost formulas for queries, update, insertion, and deletion transactions was developed in Chapter 6; they were used in the implementation of PhyDDO.

In Chapter 7 we introduced a closed noniterative formula for estimating the number of block accesses. This formula, an approximation of Yao's exact formula, has a practically negligible error and significantly reduces the computation time by eliminating the iterative loop found in Yao's formula. It also achieves a much higher accuracy than an approximation proposed by Cardenas.

Extensions of separability approach to more-than-two variable transactions were briefly discussed in Chapter 9. This was done by decomposing the transactions into a sequence of two-variable transactions. Some properties of decomposition, however, do not satisfy the conditions for separability. In the proposed methodology this violation was simply ignored. Some justification, though not complete, was given for this approximation.

The property of separability is a good property to exploit in the physical database design. To take advantage of this property, as exemplified in Chapters 3 and 8, one has to extract the maximum set of features that satisfies the conditions for separability; then, extend it using heuristics to incorporate the features not included in the separable set. To incorporate as many features as possible in the first phase, it is possible to make approximations to the cost formulas and make them separable. The cost formulas for network model databases developed by Gerritsen (see Appendix B) is a good example; these cost formulas were made separable by disregarding the possible violation of a condition for separability explained in Section 9.2. Another way to take advantage of the separability property is to design the optimizer and the join algorithms in such a way that they satisfy the conditions for separability. Some examples of the requirements are the availability of the TID intersection algorithm in manipulating multiple indexes to solve the restriction predicates and the ability of the join algorithms to take maximum advantage of the coupling effect so that either partial coupling factors or coupling factors are effective in both directions when the join index method is used.

10.2 Topics for Further Study

In many cases a large number of columns in a relation do not appear in any predicate of any transaction. An index on a column that does not appear in any predicate cannot contribute to the reduction of the access cost, but only adds its own maintenance cost. Thus, if we eliminate the indexes from consideration in a preliminary index selection step before the physical database design algorithms are invoked, we could reduce the design time significantly.

The design methodology must be extended to include more-than-two-variable transactions. A preliminary methodology was proposed in Chapter 9. Nevertheless, more elaborate schemes as well as better justification of the approximations are subject to further research.

If a relational DBMS supports additional access structures – such as the linked list structures – the design algorithm must be modified accordingly. We believe that this can be achieved by including a separate design step in the iteration loop of the design algorithm.

For network model databases, development and validation of design algorithms including nonseparable access structures are left to a further study.

The hierarchical database model employed in many existing database systems [WIE 83] were not considered in this dissertation because it violates one important assumption necessary for the propery of separability. Hierarchical database store the records of many record types closely together in a hierarchical format. Thus, records of one record type disturbs the placement of the records of other record types violating the Condition 1.3 for separability. However, relevant heuristic employed with simplifying assumptions to incorporate the theory may provide sufficient accuracy for practical purposes. More research on this possibility is left to further study.

Appendix A. Separability – An Approach to Physical Database Design

This appendix is omitted since it is available from the Proceedings of the Seventh International Conference on Very Large Databases held in Cannes, France, in September 1981.

Appendix B. Physical Design of Network Model Databases Using the Propery of Separability

This appendix is omitted since it is available from the Proceedings of the Eighth International Conference on Very Large Databases held in Mexico City, Mexico in September 1982.

Appendix C. Estimating Block Accesses in Database Organizations

This appendix is omitted since it will be published in the Communications of the ACM shortly.

Appendix D. Physical Design Algorithms for Multifile Relational Databases

This paper has been submitted for publication. For convenience all the references have been moved to the end of the thesis.

Physical Design Algorithms for Multifile Relational Databases

by

Kyu-Young Whang
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

Three algorithms for the optimal physical design of multifile relational databases are presented. Each algorithm employs different techniques of partitioning the search space to reduce the time complexity. The three design algorithms are compared with one another to validate the heuristics exploited. In an extensive test performed to determine the optimality of the design algorithms, all three found the optimal solutions in most of the cases. The time complexities of the design algorithms show a substantial improvement when compared with the approach of exhaustively searching through all possible alternatives.[3]

Categories and Subject Descriptors:

H.2.2 [Database Management]: Physical Design – *access methods*; H.3.2 [Information Storage and Retrieval]: Information Storage – *file organization*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval – *clustering, retrieval models* .

General Terms:

Algorithms, Design

Additional Key Words and Phrases:

physical database design, performance, query optimization, join methods, block accesses, index selection, selectivity

[3] This work was supported by the Defence Advanced Research Project Agency under the KBMS Project, Contract N39-82-C-0250.

Authors' current addresses: Computer Systems Laboratory ERL 416, Stanford University, Stanford, California 94305

D.1 Introduction

A good design of the physical database has a vital influence on the database performance. As such, the problem of physical database design has been given much attention in recent years [HSI 70] [CAR 75] [SCH 75] [SEV 75] [HAM 76] [YAO-a 77] [BAT 80] [GER 77] [GAM 77]. The problem concerns finding an optimal configuration of physical files and auxiliary structures —given the logical access paths that represent the interconnections among objects in the data model; the usage patterns of those paths; the organizational *characteristics of data* stored in the files as well as the various features provided by a particular database management system(DBMS). In this paper we use the term *access structures* as the features that a particular DBMS provides for the physical database design (e.g., indexes and the property of clustering). We use the term *access configuration* of a relation or of the database to mean the aggregate of access structures specified to support a relation or the entire database. Thus, the access configuration is an abstraction of the physical database.

In the past much of the research related to the physical database design concentrated on rather simple cases dealing with a single file. In a database organization that consists of multiple files, however, the data in different files have complex interrelationships and access patterns; a simple extension of single file analyses (under the assumption of independency among files) does not suffice for understanding the interactions among multiple files. Although some efforts (mainly for developing cost formulas) have been devoted to multifile cases [GER 77] [BAT 80], it is difficult to use them for the optimal design of physical databases without exhaustively searching all the possible access configurations of the database. As pointed out in [GER 77], a relevant partitioning of the entire database is necessary to make the optimal design of the physical database a practical matter.

A theory of separability was introduced in [WHA-a 81] as a formal basis for understanding the interrelationships among files. In particular, the theory proves that, given a set of join methods that satisfies a certain property called separability, the problem of designing the optimal physical database can be reduced to the subproblem of optimizing individual relations (each relation is mapped to a file) independently of one another. Once the problem has been partitioned, the techniques developed for single-file designs can be applied to solve the subproblems.

In this paper we introduce three algorithms for the optimal physical design of multifile relational databases. Our objective towards optimality in these algorithms is the minimum number of disk accesses for processing queries and update transactions. Algorithm 1 and Algorithm 2 are based on the theory of separability so that the design is performed relation by relation. These algorithms are also extended, by using heuristics, to include the join methods that are not in the separable set (we will call them nonseparable join methods). Algorithm 3 does not utilize the property of separability and designs the entire database all together. Instead, it employes a different partitioning scheme to reduce the time complexity.

The design algorithms are tested for their optimality by comparing the results they produce with the optimal solution obtained by searching exhaustively among all the possible access configurations. When a large database is involved, however, it may be practically impossible to obtain the optimal solution by an exhaustive search; in this case, the results of the three algorithms are compared to obtain a solution that is most probably the optimal.

Section D.2 introduces several key assumptions, while Section D.3 describes general classes of transactions we consider and the transaction processing methods of interest. In Section D.4 we briefly review the theory of separability. The three design algorithms are introduced in Section D.5. These algorithms have been fully implemented using a comprehensive set of cost formulas. The test results, including the accuracy of these algorithms (compared with the optimal solutions) and their performance (compared with the exhaustive-search method), are also discussed in Section D.5. More details on the development of the algorithms and the complete set of tests performed can be found in Appendices J.1 and K.

D.2 Assumptions

Several key assumptions are used throughout the paper. In principle, some assumptions are not necessary for Algorithm 3 since this algorithm does not rely on the theory of separability. But, for the purpose of comparison, we shall apply all the assumption stated in this section to the three algorithms.

We assume that the DBMS we are considering provides indexes and the clustering property of a single relation as access structures. Clustering of two or more relations, as is available in many hierarchical organizations, is not considered. We also assume that all TID (tuple identifier) manipulations can be performed in the main memory without any need to perform I/O accesses.

The database is assumed to reside on disklike devices. Physical storage space for the database is divided into units of fixed size called blocks [WIE 83]. The block is not only the unit of disk allocation, but is also the unit of transfer between main memory and disk. We assume that a block that contains tuples of a relation contains only the tuples of that relation. Furthermore, we assume that the blocks containing tuples of a relation, which comprise a file, can be accessed serially. However, the blocks do not have to be contiguous on disk.

In principle, we assume that a relation is mapped into a single file, an attribute to a column, and a tuple to a record. Accordingly, we shall use the terms *file* and *relation* interchangeably. Nor shall we make any distinction between an attribute and a column or between a tuple and a record.

Sometimes we need indexes defined for two or more attributes (multiattribute indexes). The sequence of attributes for which a multiattribute index is defined is mapped into a *virtual column.* During the design process a virtual column is considered to be independent from ordinary single-attribute columns. One exception, however, is that when a virtual column is endowed with the clustering property, its first component column should have the property too. The virtual columns are defined only for semantically appropriate sequences of attributes [WIE 79]. More detailed treatment on the virtual column can be found in Appendix J.2.

We consider only one-to-many (including one-to-one) relationships between relations. It is argued in [WHA-b 81] that many-to-many relationships between relations are less important for the optimization. Note that here we are dealing with relationships between relations based on the equality of join-attribute values; a relationship among distinct entity sets at the conceptual level is often structured with an additional intermediate relation [ELM 80].

Finally, we consider only one-variable (one-relation) or two-variable (two-relation) transactions. For a transaction of more than two variables, a heuristic approach can be employed to decompose it into a sequence of two-variable transactions. (These correspond to one-overlapping queries in [WON 76].)

D.3 Transaction Evaluation

D.3.1 Queries

The class of queries we consider is shown in Figure D-1. The conceptual meaning of this class of queries is as follows. Tuples in relation R_1 are restricted by restriction predicate P_1. Similarly, tuples in relation R_2 are restricted by predicate P_2. The resulting tuples from each relation are joined according to the join predicate $R_1.A = R_2.B$, and the result projected over the columns <list of attributes>. We call the columns that are involved in the restriction predicates *restriction columns*, and those in the join predicate *join columns*. The actual implementation of this class of queries does not have to follow the order specified above as long as it produces the same result.

$$\begin{array}{lll} \text{SELECT} & \langle\text{list of attributes}\rangle & \\ \text{FROM} & R_1, R_2 & \\ \text{WHERE} & R_1.A = R_2.B & \text{AND} \\ & P_1 & \text{AND} \\ & P_2 & \end{array}$$

Figure D-1: General Class of Queries Considered.

Query evaluation algorithms, especially for two-variable queries, have been studied in [BLA 76] and [YAO 79]. The algorithms for evaluating queries differ significantly in the way they use join methods. Before discussing the various join methods, let us define some terminology. Given a query, an index is called a *join index* if it is defined for the join column of a relation. Likewise, an index is called a restriction index if it is defined for a restriction column. We use the term *subtuple* for a tuple that has been projected over some columns. The restriction predicate in a query for each relation is decomposed into the form $Q_1 \wedge Q_2$, where Q_1 is a predicate that can be processed by using indexes, while Q_2 cannot. Q_2 must be resolved by accessing individual records. We shall call Q_1 the *index-processible predicate* and Q_2 the *residual predicate.*

Some algorithms for processing joins that are of practical importance are summarized below (see also [BLA 76] [SEL 79]):

- *Join Index Method*: This method presupposes the existence of join indexes. For each relation, the TIDs of tuples that satisfy the *index processible predicates* are obtained by manipulating the TIDs from each index involved; the resultant TIDs are stored in temporary relations R_1' and R_2'. TID pairs with the same join column values are found by scanning the join column indexes according to the order of the join column values. As they are found, each TID pair (TID_1, TID_2) is checked to determine whether TID_1 is present in R_1' and TID_2 in R_2'. If they are, the corresponding tuple in one relation, say R_1, is retrieved. When this tuple satisfies the *residual predicate* for R_1, the corresponding tuple in the other relation R_2 is retrieved and the residual.predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple of interest is constructed. (We say that the direction of the join is from R_1 to R_2.)

- *Sort-Merge Method*: The relations R_1 and R_2 are scanned – either by using restriction indexes, if there is an index-processible predicate in the query, or by scanning the relation directly – and temporary relations T_1 and T_2 are created. Restrictions, partial projections, and the initial step of sorting are performed while the relations are being initially scanned and stored in T_1 and T_2. T_1 and T_2 are sorted by the join column values. The resulting relations are scanned in parallel and the join is completed by merging matching tuples.

- *Combination of the Join Index Method and the Sort-Merge Method*: One relation, say R1, is sorted as in the sort-merge method and stored in T_1. Relation R_2 is processed as in the join index method, storing the TIDs of the tuples that satisfy the index processible predicates in R_2'. T_1 and the join column index of R_2 are scanned according to the join column values. As matching join column values are found, each TID from the join index of R_2 is checked against R_2'. If it is in R_2', the corresponding tuple in R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple is constructed.

- *Inner/Outer-Loop Join Method*: In the two join methods described above, the join is performed by scanning relations in the order of the join column values. In the inner/outer-loop join, one of the relations, say R_1, is scanned without regard to order, either by using restriction indexes or by scanning the relation directly, and, for each tuple of R_1 that satisfies predicate P_1, the tuples of relation R_2 that satisfy predicate P_2 and the join predicate are retrieved and concatenated with the tuple of R_1. The subtuples of interest are then projected upon the result. (We say the direction of the join is from R_1 to R_2.)

Let us note that, in the combination of the join index method and the sort-merge method, the operation performed on either relation is identical to that performed on one relation in the join index method or in the sort-merge method. We call the operations performed on each relation *join index method (partial)* or *sort-merge method(partial)*, respectively; whenever no confusion arises, we call these operations simply *join index method* or *sort-merge method.* According to the definitions, the join index method actually consists of two join index methods (partial) and, similarly, the sort-merge method consists of two sort-merge methods (partial).

D.3.2 Update Transactions

We assume that the updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Thus, updates are not performed on the join of two or more relations. (If they are, certain ambiguity arises on which relations to update [KEL 81].) The class of update transactions we shall be considering is shown in Figure D-2.

```
UPDATE  $R_1$
SET     $R_1$.C = <new value>
FROM    $R_1$, $R_2$
WHERE   $R_1$.A = $R_2$.B  AND
        $P_1$              AND
        $P_2$
```

Figure D-2: General Class of Update Transactions Considered.

The conceptual meaning of this class of transactions is as follows. Tuples in relation R_2 are restricted by restriction predicate P_2. Let us call the set of resulting tuples T_2. Then, the value for column C of each tuple in R_1 is changed to <new value> if the tuple satisfies the restriction predicate P_1 and has a matching tuple in T_2 according to the join predicate. In a more familiar syntax [CHA 76], the class of update transactions can be represented as in Figure D-3. The equivalence of the two representations (only for queries) has been shown in [KIM 82].

UPDATE R_1
SET R_1.C = <new value>
WHERE P_1 AND
R_1.A IN
(SELECT R_2.B
FROM R_2
WHERE P_2)

Figure D-3: An Equivalent Form of the General Class of Update Transactions.

Deletion transactions are specified in an analogous way. It is assumed that insertion transactions refer only to single relations. From now on, unless any confusion arises, we shall refer to update, deletion or insertion transactions simply as update transactions.

The update transaction in Figure D-2 can be processed just like queries except that an update operation is performed instead of concatenating and projecting out the subtuples after relevant tuples are identified. In particular, all the join methods described in Section D.3.1 can be used for update transactions as well. But, there are two constraints: 1) The sort-merge method cannot be used for the relation to be updated since it is meaningless to create a temporary sorted file for that relation. 2) When the inner/outer-loop join method is used, the direction of the join must be from the relation to be updated (R_1) to the other relation (R_2) because, if the direction were reversed, the same tuple might be updated more than once. Let us note that, although two-relation update transactions are not joins, the join predicates (ones that relate two relations) they have can be processed with the join methods defined for processing joins.

D.4 Theory of Separability

To review the design theory based on the concept of separability, we introduce a formal definition of separability, related terminology, and theorems that are relevant to relational databases. A detailed development of the theory and the proofs of the theorems can be found in [WHA-a 81].

Definition 1: The *join selectivity* of a relation R with respect to a join path JP is the ratio of the number of distinct join column values of the tuples participating in the unconditional join to the total number of the distinct join column values of R. A *join path* is a set (R_1, R_1.A, R_2, R_2.B), where R_1 and R_2 are relations participating in the join and R_1.A and R_2.B are join columns of R_1 and R_2, respectively. An *unconditional join* is a join in which the restrictions in either relation are not considered. □

Definition 2: A *connection* is a join path predefined in the schema [WIE 79]. □

Definition 3: The *coupling effect* from relation R_1 to relation R_2, with respect to each transaction, is the ratio of the number of distinct join column values of the tuples of R_1, selected according to the restriction predicate for R_1, to the total number of distinct join column values in R_1. □

If we assume that the join column values are randomly selected, the coupling effect from R_1 to R_2 is the same as the ratio of the number of distinct join column values of R_2 selected by the effect of the restriction predicate for R_1 to the total number of distinct join column values in R_2 participating in the unconditional join.

Definition 4: A *coupling factor* Cf_{12} from relation R_1 to relation R_2 with respect to a transaction is the ratio of the number of distinct join column values of R_2, selected by both the coupling effect from R_1 (through the restriction predicate for R_1) and the join selectivity of R_2, to the total number of distinct join column values in R_2. □

According to the definition, a coupling factor can be obtained by multiplying the coupling effect from R_1 to R_2 by the join selectivity of R_2. This coupling factor contains all the consequences of interactions of relations in the join operation since it includes both coupling and join filtering effects.

Definition 5: A *partial-join cost* is the part of the join cost that represents the accessing of only one relation as well as the auxiliary structures defined for that relation. □

Definition 6: A *partial-join algorithm* is a conceptual component of the algorithm of a join method whose processing cost is a partial-join cost. □

Definition 7: A set of join methods is *separable* under certain constraints, if under these constraints

- Any partial-join algorithm of a join in the set can be combined with any partial-join algorithm of any join method in the set to form a complete join method, and
- A partial-join cost of any join method in the set can be determined regardless of the partial-join algorithm used and the access configuration defined for the relation on the other side of the join. □

Theorem 1: The problem of designing the optimal access configuration of a database can be decomposed into the tasks of designing the optimal access configuration of individual relations independently of one another, if the set of join methods used by the DBMS is *separable.* □

Theorem 2: The set of join methods consisting of the join index method, the sort-merge method, and the combination method is *separable* under the constraint that, whenever the join index method is used for both relations in processing a transaction, the transaction must not have a residual predicate for at least one relation. □

A violation of the conditions for separability can occur if indexes are missing for some restriction columns on both relations participating in a join since, then, restriction predicates on both sides will contain residual predicates. It has been argued in [WHA-a 81], however, that the error in the cost estimation due to this violation is minimal. This argument has been supported by the results of the experiments to be presented in Section D.5.

Let us note that, in Theorem 2, all the join methods introduced in Section D.3 are included except for the inner/outer-loop join method. The inner/outer-loop join method is nonseparable and has to be included in the design algorithms by a heuristic extension.

D.5 Design Algorithms

In this section we introduce three algorithms for the optimal design of multifile physical databases. The most straightforward method to obtain the optimal access configuration is an exhaustive search. For even a small input situation, however, this method could be intolerably time-consuming since its time complexity increases exponentially as the size of the input situation grows. Thus, we need to partition the design steps judiciously and to develop interfaces that will minimize interactions among these steps.

The three design algorithms (Algorithms 1, 2, and 3) differ in their use of two partitioning schemes: horizontal partitioning and vertical partitioning. In the former, based on the theory of separability, the entire design is partitioned into the designs of individual relations. This scheme is also extended, by using heuristics, to include the inner/outer-loop join method which cannot be incorporated by the theory of separability. In the latter, index selection and clustering design are performed in separate steps during the process of designing a relation or the entire database. Algorithm 1 employs both horizontal and vertical partitioning; Algorithm 2 only horizontal partitioning; Algorithm 3 only vertical partitioning.

In Section D.5.1 the design algorithms are described in detail. Their time complexities are presented in Section D.5.2. Validation of the design algorithms is discussed in Section D.5.3.

D.5.1 Three Algorithms

The three algorithms are illustrated in Figures D-4, D-5, and D-6, respectively. The input information for and the output results from the design algorithms are as follows:

Input:

- Usage information: A set of various queries and update transactions with their frequencies.
- Data Characteristics: The logical schema including connections; (for each relation in the database) cardinality, blocking factor, index blocking factors and selectivities of all columns, relationships with respect to connections, join selectivities with respect to connections.
- Derived inputs: Coupling factors with respect to individual two-variable transactions. (These are derived from the data characteristics and the restriction predicates in the transactions.)

Output:

- The optimal access configuration of the database, which consists of the optimal position of the clustering column and the optimal index set for each relation.
- The optimal join method for each two-variable transaction.

D.5.1.1 Algorithm 1

The design is performed in two phases: Phase 1 and Phase 2. These two phases are iterated until the refinement through the loop becomes negligible (1 %). In Phase 1, based on the theory of separability, the access configuration is designed relation by relation independently of one another using only the join methods in the separable set—the join index method, the sort-merge method, and the combination method. Phase 1 is further divided into two steps: the Index Selection Step and the Clustering Design Step. In the Index Selection Step an optimal index set is chosen given the clustering column position determined in the Clustering Design Step of the last iteration. (In the first iteration, there is no clustering column initially.) In the Clustering Design Step, an optimal clustering column position is chosen given the index set determined in the Index Selection Step.

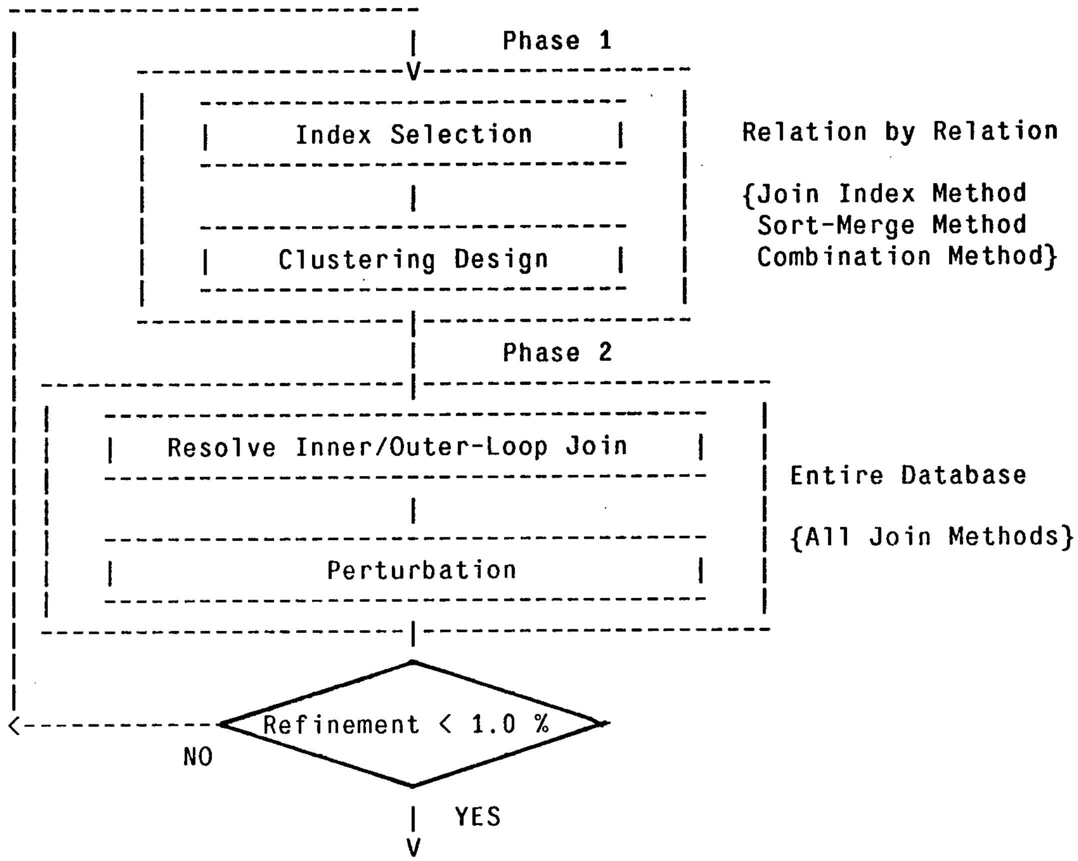


Figure D-4: Algorithm 1 for the Optimal Design of Physical Databases.

Before introducing the details for these steps, we define the function EVALCOST-1 as follows:

Function EVALCOST-1

Input:

- Access configuration of the relation being considered.
- Set of transactions that are to be processed in Phase 1 using the inner/outer-loop join method and the direction of the join for each transaction in the set. (These transactions are identified in Phase 2 of the previous iteration.)

Output:

- Total cost of the relation.

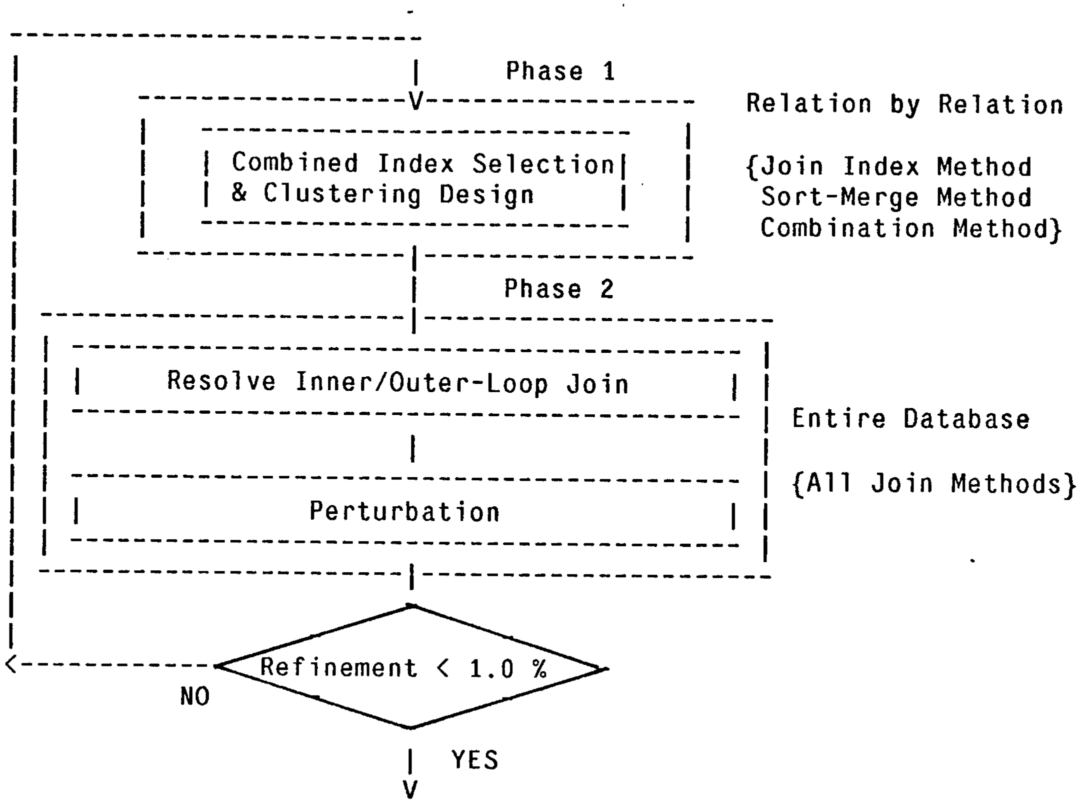


Figure D-5: Algorithm 2 for the Optimal Design of Physical Databases.

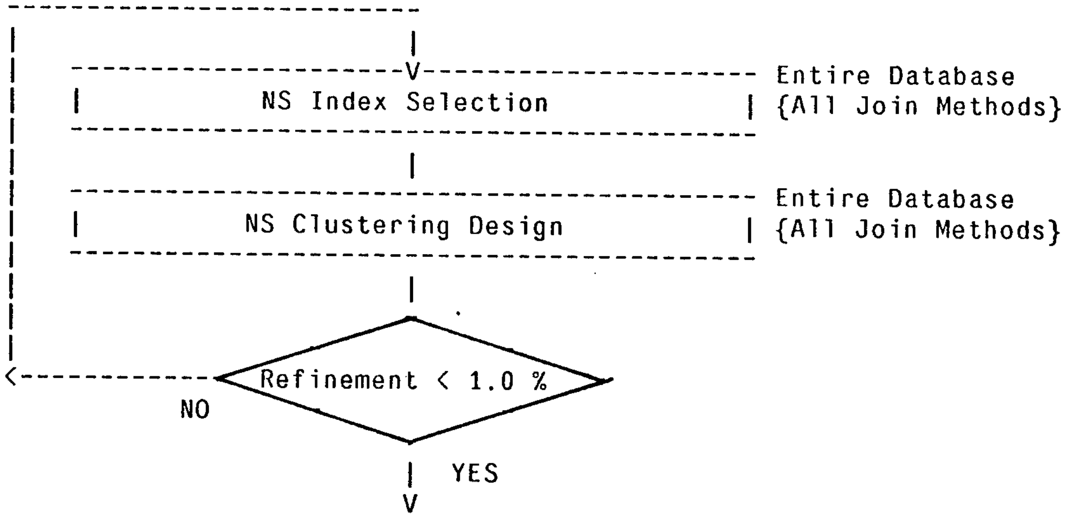


Figure D-6: Algorithm 3 for the Optimal Design of Physical Databases.

(In the input specification of this function as well as the functions or algorithms introduced later, the global input information introduced at the beginning of this section is assumed implicit unless stated otherwise.)

The total cost of a relation is obtained by summing up the costs of single-relation transactions and the partial-join costs of two-relation transactions that refer to the relation. The cost of each transaction must be multiplied by its frequency. For each partial-join, the best partial-join algorithm is selected and its cost calculated. However, if the transaction is supposed to be processed by the inner/outer-loop join method, that method will be used unconditionally according to the join direction specified because the inner/outer-loop join method cannot be treated uniformly with separable join methods in Phase 1 due to its nonseparable nature.)

Using the function EVALCOST-1 defined above, the algorithm for index selection is described as follows:

Index Selection Step

Input:

- Clustering column position for each relation
- Set of transactions that are to be processed using the inner/outer-loop join method and the direction of the join for each transaction in the set.

Output:

- The optimal index set for each relation with respect to the input information.

Algorithm:

1. Pick one relation and start with an access configuration having a full index set.
2. Try to drop one index at a time and apply EVALCOST-1 to the resulting access configuration to find the index that yields the maximum cost benefit when dropped.
3. Drop that index.
4. Repeat Steps 2 and 3 until there is no further reduction in the cost.
5. Try to drop two indexes at a time and apply EVALCOST-1 to the resulting access configuration to find the index pair that yields the maximum cost benefit when dropped.

6. Drop that pair.

7. Repeat Steps 5 and 6 until there is no further reduction in the cost.

8. Repeat Steps 5, 6, and 7 with three indexes, four indexes, ..., up to k (k must be predefined) indexes at a time.

9. Repeat the entire procedure for every relation in the database.

The variable k, the maximum number of indexes that are dropped together at a time, must be supplied to the algorithm by the user. We believe, however, that k=2 suffices in most practical cases. In fact, in all the tests performed to validate the design algorithms, the maximum value of k actually exploited was 1 (i.e., no improvement was observed with larger values of k).

The index selection algorithm presented here bears some resemblance to the one introduced by Hammer and Chan [HAM 76], but it uses the Drop Heuristic [FEL 66] instead of the ADD Heuristic [KUE 63]. The Drop Heuristic attempts to obtain an optimal solution by incrementally dropping indexes starting with a full index set. On the other hand, the ADD Heuristic adds indexes incrementally starting from an initial configuration without any index to reach an optimal solution. Since we are pursuing a heuristic approach for index selection, the actual result is suboptimal. However, in most of the cases we tested, the algorithm found optimal solutions. More details on the index selection algorithm, its validation, and the advantage of the Drop Heuristic over the ADD Heuristic will be presented in Appendix F.

The Clustering Design Step comes next in Phase 1.

Clustering Design Step

Input:

- Index set for each relation determined in the Index Selection Step.
- Set of transactions that are to be processed using the inner/outer-loop join method, and the directions of the join for each transaction in the set.

Output:

- Optimal position of the clustering column for each relation with respect to the input information.

Algorithm:

1. Select one relation.
2. Assign the clustering property to one column of the relation.
3. Apply EVALCOST-1 to the resulting access configuration.
4. Shift the clustering property to another column of the relation and repeat Steps 2 and 3.
5. Repeat Step 4 until all the columns of the relation have been considered, including the configuration having no clustering column is also considered. Then determine the one that gives the minimal cost as the clustering column (or none).

In Substep 2 the clustering property accompanies an index if the column has not been assigned one in the Index Selection Step. This strategy slightly enhances the accuracy of the design algorithms. More details on this strategy as well as other strategies enhancing the accuracy can be found in Appendix J.1.

The clustering design algorithm amounts to an enumeration of all possible alternatives. However, because of the restriction that a relation can have at most one clustering column, the time complexity is only linear on the number of columns in the relation. When a virtual column is involved, there could be more than one clustering column in a relation since the first component column of a virtual column that is clustering is itself a clustering column. But, since the two columns are tightly interlocked, the time complexity is still linear on the number of columns (now including virtual columns) in the relation.

In Phase 2 the design algorithm is extended to include the inner/outer-loop join method. Since the inner/outer-loop join method is nonseparable, it cannot be incorporated in Phase 1. Instead, a separate step (Resolve Inner/Outer-Loop Join Step) is attached to take a corrective action. Given the access configuration from Phase 1, for each two-relation transaction, the best join method is selected. If the inner/outer-loop join method happens to be the best one, it is remembered that the transaction be processed by the inner/outer-loop join method in Phase 1 of the next iteration. Also remembered is the direction of the join. To describe the algorithm for the Resolve Inner/Outer-Loop Join Step, we define the function EVALCOST-2.

Function EVALCOST-2

Input:

- Access configuration of the entire database.

Output:

- Total cost of the database.

Side Effect:

- Two-relation transactions that use the inner/outer-loop join method are marked, and their join directions recorded.

The total cost of the database is obtained by summing up the costs of all transactions multiplied by their respective frequencies. For each two-relation transaction, the best join method (including the inner/outer-loop join method) is selected and its cost calculated. As a side effect, if the best join method for a transaction is the inner/outer-loop join method, a reminder is attached to the transaction that it must be processed by the inner/outer-loop join method in Phase 1 of the next iteration. This reminder is one of the elements that interfaces Phase 1 and Phase 2 conveying information from one phase to another.

The following is the algorithm for Resolve Inner/Outer-Loop Join Step:

Inner/Outer-Loop Join Step

Input:

- The access configuration of the database from Phase 1.

Output:

- Set of transactions to be processed by the inner/outer-joop join method and the direction of the join for each transaction in the set.

Algorithm:

1. Apply EVALCOST-2 once. The desired output-will be obtained by the side effects of EVALCOST-2.

The second step of Phase 2 is the Perturbation Step. This step eliminates snags in the design process incurred by some anomalies. One anomaly is due to the peculiar characteristics of update transactions; that is, in processing an update transaction, the join index always remains after Phase 1 during the first iteration because the join index method is the only one available to resolve the join predicate for the relation being updated. (The sort-merge method is not allowed for the relation to be updated; the inner/outer-loop join method cannot be used in Phase 1 of the first iteration.) A problem arises in the Resolve Inner/Outer-Loop Join Step when the inner/outer-loop join is costlier than the join index method, but less costly if the maintenance (update) cost of the join index is incorporated. In this situation it would be more beneficial to use the inner/outer-loop join method and drop the join index. But, since the Inner/Outer-Loop Join Step does not incorporate the index maintenance cost, the algorithm finds the join index method less costly and lets the join index stay. Hence, we may never have a chance to drop the index. Simply adding the maintenance cost to that of the join index method will not work since the maintenance cost of an index must be shared by all transactions accessing that index. Therefore, in the Perturbation step, we try to drop the join index and compare the total transaction processing costs before and after the change. If the change proves to be beneficial, the join index is actually dropped.

Another anomaly occurs because we consider the inner/outer-loop join method separately from the other join methods. Sometimes the presence of an index favors performing the inner/outer-loop join in a certain direction. Dropping that index and reversing the direction of the inner/outer-loop join, however, may be more beneficial. But, it is impossible to consider this alternative in the Inner/Outer-Loop Join Step since that step is not allowed to change the access configuration. To solve this problem, in the Perturbation Step, we also try to drop an arbitrary index (as well as join indexes) and make the change permanent if it reduces the cost.

We generalize this concept and try to *add* an index as well as to *drop* one. Here, the algorithm for the Perturbation Step follows:

Perturbation Step:

Input:

- Access configuration from Phase 1.
- Total cost of the database obtained in the Inner/Outer-Loop Join Step.

Output:

- Modified access configuration of the database.

Algorithm:

1. Pick a column in the database. Try to drop the index if the column has one; otherwise add one.
2. Obtain the total cost of the database using EVALCOST-2. If the change reduces the cost, make it permanent.
3. Repeat Steps 1 and 2 for every column in the database.

We note that the Perturbation Step is supposed to accomplish a minor revision in the current access configuration to eliminate the snags that obstruct a smooth flow of the design process. Thus, only a small number of columns will be affected by the Perturbation Step; the affected columns must be sparsely scattered, and relatively independent of one another. Accordingly, dropping or adding two or more indexes together is excluded from consideration. For the same reason, an arbitrary order is chosen in considering the columns.

D.5.1.2 Algorithm 2

Algorithm 2 is almost identical to Algorithm 1 except that the two steps in Phase 1 are combined in one design step: the Combined Index Selection and Clustering Design Step (Combined Step). The algorithm is described below:

Combined Step:

Input:

- Set of transactions that are to be processed using the inner/outer-loop join method and the direction of the join for each transaction in the set.

Output:

- Optimal access configuration for each relation with respect to the input information.

Algorithm:

1. For each clustering column position in a relation, perform index selection
2. Save the best configuration.

As we shall see in Section D.5.2, the time complexity of Algorithm 2 is greater than that of Algorithm 1. The purpose of merging two steps in Phase 1 into one despite the increase in time complexity is to validate the heuristic of separating the two steps of Phase 1 (vertical partitioning) in Algorithm 1. This can be done by comparing the results from Algorithm 1 and Algorithm 2: since Algorithm 2 does not use vertical partitioning, if the results from the two algorithms are always identical, we can conclude that the deviations from the optimal solution that may exist have not been incurred by vertical partitioning.

D.5.1.3 Algorithm 3

Algorithm 3 is different from Algorithms 1 and 2 in that it does not employ horizontal partitioning and, accordingly, does not rely on the property of separability. The algorithm consists of one phase which, in turn, is decomposed into two steps: the NS Index Selection Step and the NS Clustering Design Step (the prefix NS stands for "nonseparable"). The two steps design the access configuration of the entire database all together rather than relation by relation. All available join methods are incorporated. The algorithms are described below:

NS Index Selection Step

Input:

- Clustering column positions determined in the NS Clustering Design Step of the last iteration.

Output:

- Optimal index set of entire database with respect to the given clustering column positions.

Algorithm:

1. Identical to the Index Selection Step except that the index set is designed for the entire database at the same time and using the function EVALCOST-2.

NS Clustering Design Step

Input:

- Index set of the database determined in the NS Index Selection Step.

Output:

- Optimal positions of the clustering columns with respect to the given index set.

Algorithm:

1. Start with an access configuration having no clustering columns.

2. Try to assign the clustering property to one column in the database at a time. Applying EVALCOST-2, find the column that yields the maximum cost benefit.

3. Assign the clustering property to that column.

4. Repeat Steps 2 and 3 with the constraint that one relation can have at most one clustering column until there is no further reduction in the cost.

5. Starting with the access configuration from Step 4, try to assign the clustering property to two columns in the database at a time. One relation can have at most one clustering column. Applying EVALCOST-2, find the pair that yields the maximum cost benefit.

6. Assign the clustering property to that pair.

7. Repeat Steps 5 and 6 until there is no reduction in the cost.

8. Repeat Steps 5, 6, and 7 with three columns, four columns, ..., up to k columns (k must be predefined) at a time.

As shown in Section D.5.2, the time complexity of Algorithm 3 is much greater than those of Algorithm 1 and 2. Yet, Algorithm 3 is necessary to validate the horizontal partitioning strategy. Since horizontal partitioning is based on theory, it is not a heuristic if the set of join methods available is separable. In Algorithms 1 and 2, however, horizontal partitioning is used even though the set of join methods considered is not separable because of the inner/outer-loop join method. This is done by using only the separable set of join methods in Phase 1 that excludes the inner/outer-loop join method, and by adding Phase 2 to incorporate the inner/outer-loop join method. Clearly, a heuristic is involved in this procedure, and it oughtto be validated. As Algorithm 3 does not adopt horizontal partitioning, the heuristic can be validated by comparing the results from Algorithm 1 with that from Algorithm 3.

D.5.2 Time Complexities of Design Algorithms

In this section we discuss the time complexities of the three design algorithms. Time complexities are estimated in terms of the number of calls to the cost evaluator (EVALCOST-1 or EVALCOST-2) which is the costliest operation in the design process. The actual performance measured in the test runs is summarized in Table 1 in Section D.5.3.

The overall time complexity of Algorithm 1 is $O(t \times v^{k+1}) + O(t \times c)$, where t is the number of transactions specified in the usage information, v the average number of columns in a relation, c the number of columns in the entire database, and k the maximum number of columns considered together in the Index Selection Step. Phase 1 contributes to the first term in the complexity; Phase 2 to the second.

Among the two design steps in Phase 1, the Clustering Design Step has a time complexity $O(t \times v)$ which is dominated by that of the Index Selection Step. In the Index Selection Step EVALCOST-1 is called for every k-combination of columns of the relation being considered and for every transaction that refers to the relation. This contributes the order of $(s/r) \times t \times v^k$, where r is the number of relations in the database and s is the average number of relations that a transaction refers to. (Thus, (s/r) represents the average ratio of the number of transactions referring to a particular relation to the total number of transactions.) This procedure is repeated until there is no further reduction in the cost (the number of repetitions is proportional to v). Since the entire procedure is repeated for every relation, the overall time complexity of Phase 1 is $O(t \times v^{k+1})$ if we assume that s is relatively fixed. More detailed derivation of the time complexity of the Index Selection Step can be found in Appendix J.3.

In Phase 2, the Resolve Inner/Outer-Loop Join Step requires only one call to EVALCOST-2; thus, it is dominated by the Perturbation Step. The Perturbation Step calls EVALCOST-2 for every column in the database and for every transaction in the usage. As a result, the time complexity of this step is $O(t \times c)$. Let us note that if v, the average number of columns in a relation, is relatively fixed, the time complexity of Algorithm 1 is linear on c, the total number of columns in the database.

The time complexity of Algorithm 2 is almost identical to, but slightly greater than, that of Algorithm 1. Since the index selection is repeated for every possible clustering column position, the time complexity of Phase 1 should be multiplied by v, resulting in $O(t \times v^{k+2})$. Thus, the overall time complexity becomes $O(t \times v^{k+2}) + O(t \times c)$.

The time complexity of Algorithm 3 is estimated to be $O(t \times c^{k+1})$. Both the NS Index Selection Step and NS Clustering Design Step contribute the same order of complexity. The time complexities of both steps can be obtained by a derivation similar to the one used for the Index Selection Step. The only difference is that v, the average number of columns in a relation, is replaced by c, the number of columns in the database, since the entire database is designed all together.

In summary, Algorithm 1 is the most efficient since it employs both horizontal partitioning and vertical partitioning. Algorithm 2 is slightly more complex than Algorithm 1 but faster than Algorithm 3. Although the formula for the time complexity of Phase 1 of Algorithm 1 resembles that of Algorithm 3, the former is significantly faster in most practical situations since c is much greater than v (c/v = number of relations in the database). Yet, all three algorithms are much more efficient compared with the Exhaustive-Search Method whose time complexity is $O(t \times (v+1)^r \times 2^c)$. (See Appendix J.3 for the derivation.)

D.5.3 Validation of Design Algorithms

An important task in developing heuristic algorithms is their validation. Because physical database design is such a complex problem, finding mathematical worst-case bounds on the deviations from the optimality (we shall simply call them *deviations*) of the solutions produced by heuristic algorithms is virtually impossible. Consequently, we have to rely on empirical test results of the algorithms for their validation. In particular, we try to measure the deviations of the heuristic solutions from the optimal ones for various test input situations. In many cases, however, identifying the optimal solution itself is a difficult, often impossible, task. For simple situations optimal solutions can be obtained by exhaustively searching through all the possible alternatives. For more complex situations, however, an exhaustive search is practically prohibited by its exponentially increasing complexity. For example, an input situation consisting of twelve columns in five relations and twelve transactions generates 1.66 million possible access configurations. (It took a DECSYSTEM-20 26 hours of CPU time to find the optimal solution.) We have the following strategy for the validation of the design algorithms:

1. For simple situations the optimal solutions are obtained by exhaustively searching through all the possible access configurations. The optimal solutions are subsequently compared with the solutions generated by the design algorithms.

2. For more complex situations the solutions from Algorithms 1, 2, and 3 are considered to be optimal if all three are identical.

The second rule is based on the discussions in Sections D.5.1.2 and D.5.1.3. In essence, the rule is valid because it is very unlikely that different sources of deviations (i.e., heuristics) can cause exactly the same deviations.

The three design algorithms were tested with 21 different input situations (seven different schemas with three variations of usage inputs), and the results are summarized in Table 1. In the first column the first digit of the input situation number represents the schema, and the second the usage input. In the description, r stands for the number of relations, c the number of columns in the database, and t the number of transactions in the usage input. The CPU time shows the performance of the algorithms when run in a DECSYSTEM-20. Marked by "*" are the situations in which any deviation occurred. In most situations tested all three algorithms produced optimal solutions. Even in the situations that produced nonoptimal solutions, the deviations were far from being significant. (Algorithm 1 yielded 3.1% of deviation in Situation 50 and 6.6% in Situation 42; Algorithm 3 yielded 6.6% in Situation 42. These situations are fully analyzed in J.4.)

As we can see in Table 1, an exhaustive search takes excessive computation time even with small input situations; in comparison, all three algorithms are far more efficient without significant loss of accuracy. For a very large database (such as the one consisting of 250 relations and 5000 columns),

Table 1: Performance and Accuracy of Design Algorithms

Input Situation	Description	CPU time(s:seconds;m:minutes;h:hours;y:years)						
		Algorithm1		Algorithm2		Algorithm3		Ex.Search
10	2r, 6c, 7t	0.86s		1.25s		1.83s		26.91s
20	4r, 9c,10t	1.23s		1.48s		5.41s		36.75m
30	4r, 12c,12t	2.09s		3.44s		10.51s		13.93h
40	4r, 11c,13t	2.04s		2.73s		6.62s		3.65h
50	5r, 12c,12t	2.32s	*	4.89s		12.51s		25.85h
60	4r, 11c,15t	2.63s		3.54s		13.93s		8.52h
70	16r,110c,81t	1.63m		4.80m		2.00h		10^{35} y†
11	2r, 6c, 7t	0.84s		1.26s		1.81s		26.46s
21	4r, 9c,10t	1.35s		1.67s		5.91s		42.83m
31	4r, 12c,12t	2.17s		3.43s		10.67s		14.00h
41	4r, 11c,13t	1.42s		1.88s		9.90s		3.62h
51	5r, 12c,12t	3.54s		5.00s		13.13s		26.63h
61	4r, 11c,15t	2.71s		3.74s		21.51s		8.04h
71	16r,110c,81t	2.13m		4.60m		2.02h		10^{35} y†
12	2r, 6c, 5t	0.57s		0.86s		1.23s		17.23s
22	4r, 9c, 5t	0.43s		0.55s		1.50s		10.43m
32	4r, 12c, 6t	1.08s		1.73s		4.65s		5.95h
42	4r, 11c, 6t	0.25s	*	0.43s		0.95s	*	29.95m
52	5r, 12c, 8t	1.49s		2.41s		5.04s		9.95h
62	4r, 11c, 6t	1.23s		1.81s		3.72s		2.12h
72	16r,110c,38t	21.76s		1.77m		24.40m		10^{35} y†

† Values are estimated.
* Situations that produced nonoptimal solutions.

however, even Algorithm 3 can become intolerably time-consuming. In these cases Algorithms 1 and 2, which are based on the theory of separability, are the only algorithms applicable. When a very large database is involved, the entire physical database design somehow has to be partitioned to achieve a reasonable performance in the design process. The theory of separability provides a theoretical background to achieve this goal: it provides a clean partitioning and allows us to avoid overreliance on heuristics which are often difficult to validate.

D.6 Summary and Conclusion

Three algorithms have been presented for the optimal physical design of multifile relational databases. Each algorithm employs different techniques for partitioning the search space to reduce the time complexity and is compared to the other algorithms to validate the heuristics involved. All three algorithms are far more efficient without significant loss of accuracy than the approach of exhaustively searching through all possible alternatives.

It has been emphasized that the entire design has to be properly partitioned when a very large database is considered. The theory of separability provides a theoretical basis for this partitioning and allows us to avoid overreliance on heuristics which are often difficult to justify. (Previous work [WHA-b 82] has shown that the theory can also be applied to network model databases.)

The primary contribution of this paper is to pioneer the research on the automatic design of multifile physical databases. The multifile physical design problem has long been considered "difficult" [LUM 78]. Consequently, to the extent of the author's knowledge, no other successfully tested algorithm has been reported. (One was presented in [SCH 79], but the issue on its validity has not been addressed.) We believe that our approach can enable substantial progress to be made towards the optimal design of multifile physical databases.

Acknowledgment

This work was supported by the Defense Advanced Research Project Agency, under the KBMS project, Contract Number N39-82-C-0250.

Appendix E. Transaction-Processing Costs in Relational Databases

This paper has been submitted for publication. For convenience all the references have been moved to the end of the thesis.

Transaction Processing Costs in Relational Database Systems

by

Kyu-Young Whang
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

Accurate estimation of transaction processing costs is important for both query optimization and physical database design. Although cost formulas have been partially developed in many articles, it appears that in no place a comprehensive set of cost formulas have been introduced. In this paper a complete set of formulas for estimating the costs of queries, update, insertion, and deletion transactions is developed. The costs are measured in terms of the number of disk accesses. Although the cost formulas are based on the specific model proposed, the underlying ideas can be easily extended to other models of database systems.[4]

E.1 Introduction

Since the relational model of data was introduced by Codd [COD 70], many relational database management systems (DBMS) have been implemented [KIM 79]. A standadizing effort on relational systems is summarized in [BRO 82]. One of the important characteristics of most relational DBMS's is the *optimizer* which automatically translates the transactions expressed in a nonprocedural language to an optimal sequence of access operations to evaluate the transactions. In these systems the user need not know the physical structure of the database. Instead, the optimizer estimates the cost of each possible alternative for processing the transaction based on the given physical structure of the database and figures out the minimum-cost sequence of access operations. This procedure has been generally known as *query optimization.* Various algorithms for query optimization have been extensively studied in [SMI 75] [PEC 75] [GOT 75] [BLA 76] [YAO 79] [SEL 79].

[4] This work was supported by the Defense Advanced Research Project Agency under the KBMS Project, Contract N39-82-C-0250.

Author's current address: Computer Systems Laboratory ERL 416, Stanford University, Stanford, California 94305

A related issue that has a critical effect on the database performance is physical database design. The problem addresses the optimal configuration of the physical database so that the minimum average transaction processing cost is optained [SCH 75] [HAM 76] [KAT 80] [SCH 79] [WHA-a 81]. The information on the physical database will be used by the optimizer at run time to estimate the costs of processing transactions.

In both problems – query optimization and physical database design – an accurate cost model is needed to predict the costs of transaction-processing alternatives. Various cost models have been developed in [HSI 70] [CAR 75] [SEV 75] [BLA 76] [YAO-a 77] [GER 77] [YAO 79] [SEL 79] [SCH 81]. But, in many of them, cost formulas are either only partially developed – either only for queires or only for update transactions – or too much abstracted to be useful in practical systems.

The purpose of this paper is to introduce a comprehensive set of formulas for estimating the costs of processing queries, update, insertion, and deletion transactions in relational database systems that support the clustering (records are clustered if they are stored in the order of values of a column) and indexes. The costs are measured in terms of the number of disk accesses needed for processing transactions.

In Section E.2 we introduce key assumptions and the model of the storage structure. Section E.3 describes the general class of transactions and the transaction processing methods that we consider. Terminology is defined in Section E.4 to help understand the interactions among different relations in evaluating transaction-processing costs. Elementary cost formulas are developed in Section E.5. Finally, the transaction-processing costs are developed in Section E.6 as composites of elementary cost formulas.

E.2 Assumptions and the Model of Storage Structure

E.2.1 General Assumptions

The database is assumed to reside on disk-like devices. Physical storage space for the database is divided into fixed-size units called blocks [WIE 83]. The block not only is the unit of disk allocation but also is the unit of transfer between main memory and disk. We assume that a block that contains tuples of a relation contains only the tuples of that relation. For simplicity, we assume that a relation is mapped into a single file. Accordingly, from now on, we will use the terms *file* and *relation* interchangeably; nor shall we make any distinction between an attribute and a column or between a tuple and a record.

We assume that no block access will be incurred if the next tuple (or index entry) to be accessed resides in the same block as that of the current tuple (or index entry); otherwise, a new block access is necessary. We also assume that all TID (tuple identifier) manipulations can be performed in main memory without any need for I/O accesses.

We consider only one-to-many (including one-to-one) relationships between relations. It is argued in [WHA-b 81] that many-to-many relationships between relations are less important for the optimization purpose. Note that here we are dealing with relationships in relational representations based on the equality of join-attribute values; a relationship among distinct entity sets at the conceptual level is often structured with an additional intermediate relation [ELM 80].

Finally, we consider only one-variable (one-relation) or two-variable (two-relation) transactions. The cost for a transaction of more than two variables can be obtained by decomposing it into a sequence of two-variable transactions. (This corresponds to one-overlapping queries in [WON 76].)

E.2.2 Storage Structure of the Data File

A relation can be sorted according to the order of certain column values (say column A). We say that column A is the *clustering column* or that column A has the *clustering property.* A relation can have only one clustering column since clustering requires a specific order for storing tuples.

In each block of a file there are *slots* that contain the byte offsets of data tuples stored in that block. The addresses of these slots are called *tuple identifiers (TID),* and the tuples are located by TIDs. The TID slots provide a level of indirection so that the TIDs remain unchanged even though the tuples are shuffled in the block according to update, insertion, or deletion operations. When a new tuple is inserted, a TID slot that is the nearest to the desired place is chosen. This strategy saves the cost of shuffling data tuples and changing pointers to them. Even though this strategy may not keep the file strictly sorted according to the clustering column values, it keeps the tuples having close values near one another.

E.2.3 Storage Structure of the Index

A B^+-tree index [COM 79] can be defined for a column of a relation. The leaf-level of the index consists of (key, TID) pairs for every tuple in that relation and the leaf-level blocks are chained so that the index can be scanned without traversing the index tree. Entries having the same key value are ordered by TID. An index is called a *clustering index* if it is defined for a clustering column. We assume that no block is fetched more than once when tuples are retrieved by sequentially scanning the clustering index. When index entries are inserted or deleted, we assume that, compared with the accesses to the index blocks themselves, splits or mergers of index blocks are rather infrequent because these happen only when an index block is either completely full or empty; hence we assume that modifications are mainly done on the leaf-level blocks.

E.3 Transaction Evaluation

E.3.1 Queries

The class of queries we consider is shown in Figure E-1. The conceptual meaning of this class of queries is as follows. Tuples in relation R_1 are restricted by restriction predicate P_1. Similarly, tuples in relation R_2 are restricted by predicate P_2. The resulting tuples from each relation are joined according to the join predicate $R_1.A = R_2.B$, and the result is projected over the columns <list of attributes>. We call the columns that are involved in the restriction predicates *restriction columns*, and those in the join predicate *join columns*. The actual implementation of this class of queries does not have to follow the order specified above as long as it produces the same result.

$$\begin{array}{lll} \text{SELECT} & \langle\text{list of attributes}\rangle & \\ \text{FROM} & R_1, R_2 & \\ \text{WHERE} & R_1.A = R_2.B & \text{AND} \\ & P_1 & \text{AND} \\ & P_2 & \end{array}$$

Figure E-1: General Class of Queries Considered.

Query evaluation algorithms, especially for two-variable queries, have been studied in [BLA 76] and [YAO 79]. The algorithms for evaluating queries differ significantly in the way they use join methods. Before discussing the various join methods, let us define some terminology. Given a query, an index is called a *join index* if it is defined for the join column of a relation. Likewise, an index is called a restriction index if it is defined for a restriction column. We use the term *subtuple* for a tuple that has been projected over some columns. The restriction predicate in a query for each relation is decomposed into the form $Q_1 \wedge Q_2$, where Q_1 is a predicate that can be processed by using indexes, while Q_2 cannot. Q_2 must be resolved by accessing individual records. We shall call Q_1 the *index-processible predicate* and Q_2 the *residual predicate.*

Some algorithms for processing joins that are of practical importance are summarized below (see also [BLA 76] [SEL 79]):

- *Join Index Method*: This method presupposes the existence of join indexes. For each relation, the TIDs of tuples that satisfy the *index processible predicates* are obtained by manipulating the TIDs from each index involved; the resultant TIDs are stored in temporary relations R_1' and R_2'. TID pairs with the same join column values are found by scanning the join column indexes according to the order of the join column values. As they are found, each TID pair (TID_1, TID_2) is checked to determine whether TID_1 is present in R_1' and TID_2 in R_2'. If they are, the corresponding tuple in one relation, say R_1, is retrieved. When this tuple satisfies the *residual predicate* for R_1, the corresponding tuple in the other relation R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple of interest is constructed. (We say that the direction of the join is from R_1 to R_2.)

- *Sort-Merge Method*: The relations R_1 and R_2 are scanned – either by using restriction indexes, if there is an index-processible predicate in the query, or by scanning the relation directly – and temporary relations T_1 and T_2 are created. Restrictions, partial projections, and the initial step of sorting are performed while the relations are being initially scanned and stored in T_1 and T_2. T_1 and T_2 are sorted by the join column values. The resulting relations are scanned in parallel and the join is completed by merging matching tuples.

- *Combination of the Join Index Method and the Sort-Merge Method*: One relation, say R1, is sorted as in the sort-merge method and stored in T_1. Relation R_2 is processed as in the join index method, storing the TIDs of the tuples that satisfy the index processible predicates in R_2'. T_1 and the join column index of R_2 are scanned according to the join column values. As matching join column values are found, each TID from the join index of R_2 is checked against R_2'. If it is in R_2', the corresponding tuple in R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple is constructed.

- *Inner/Outer-Loop Join Method*: In the two join methods described above, the join is performed by scanning relations in the order of the join column values. In the inner/outer-loop join, one of the relations, say R_1, is scanned without regard to order, either by using restriction indexes or by scanning the relation directly, and, for each tuple of R_1 that satisfies predicate P_1, the tuples of relation R_2 that satisfy predicate P_2 and the join predicate are retrieved and concatenated with the tuple of R_1. The subtuples of interest are then projected upon the result. (We say the direction of the join is from R_1 to R_2.)

Let us note that, in the combination of the join index method and the sort-merge method, the operation performed on either relation is identical to that performed on one relation in the join index method or in the sort-merge method. We call the operations performed on each relation *join index method (partial)* or *sort-merge method (partial)*, respectively; whenever no confusion arises, we call these operations simply *join index method* or *sort-merge method.* According to the definitions, the join index method actually consists of two join index methods (partial) and, similarly, the sort-merge method consists of two sort-merge methods (partial).

E.3.2 Update Transactions

We assume that the updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Thus, updates are not performed on the join of two or more relations. (If they are, ambiguity may arise on which relations to update [KEL 81].) The class of update transactions we shall consider iş shown in Figure E-2.

```
UPDATE R_1
SET    R_1.C = <new value>
FROM   R_1, R_2
WHERE  R_1.A = R_2.B AND
       P_1           AND
       P_2
```

Figure E-2: General Class of Update Transactions Considered.

The conceptual meaning of this class of transactions is as follows. Tuples in relation R_2 are restricted by restriction predicate P_2. Let us call the set of resulting tuples T_2. Then, the value for column C of each tuple in R_1 is changed to <new value> if the tuple satisfies the restriction predicate P_1 and has a matching tuple in T_2 according to the join predicate. In a more familiar syntax [CHA 76], the class of update transactions can be represented as in Figure E-3. The equivalence of the two representations (only for queries) has been shown in [KIM 82].

```
UPDATE R_1
SET    R_1.C = <new value>
WHERE  P_1     AND
       R_1.A   IN
        (SELECT R_2.B
         FROM   R_2
         WHERE  P_2 )
```

Figure E-3: An Equivalent Form of the General Class of Update Transactions.

Deletion transactions are specified in an analogous way. It is assumed that insertion transactions refer only to single relations. From now on, unless confusion may occur, we shall refer to update, deletion or insertion transactions simply as update transactions.

The update transaction in Figure E-2 can be processed just like queries except that an update operation is performed instead of concatenating and projecting out the subtuples after relevant tuples are identified. In particular, all the join methods described in Section E.3.1 can be used for update transactions as well. But, there are two constraints: 1) The sort-merge method cannot be used for the relation to be updated since it is meaningless to create a temporary sorted file for that relation. 2) When the inner/outer-loop join method is used, the direction of the join must be from the relation to be updated (R_1) to the other relation (R_2) because, if the direction were reversed, the same tuple might be updated more than once. Let us note that, although two-relation update transactions are not joins, the join predicates—which relate two relations— they have can be processed with the join methods defined for processing joins.

E.4 Terminology

E.4.1 Notation

R	: A relation.
Other(R)	: The relation to be joined with R.
C	: A column.
n_R	: Number of tuples in relation R (cardinality).
p_R	: Blocking factor of relation R.
L_C	: Blocking factor of the index for column C.
F_C	: Selectivity of column C or its index
cc	: Subscript for the clustering column.
m_R	: Number of blocks in relation R, which is equal to n_R/p_R.
im_C	: Number of blocks that the index for column C occupies.
t	: A transaction
$H_{t,R}$	: Projection factor of transaction t on relation R.

E.4.2 Definition of Terms

Definition 1: The *join selectivity* $JSEL_{R,JP}$ of a relation R with respect to a join path JP is the ratio of the number of distinct join column values of the tuples participating in the unconditional join to the total number of distinct join column values of R. A *join path* is a set (R_1, R_1.A, R_2, R_2.B), where R_1 and R_2 are relations participating in the join and R_1.A and R_2.B are join columns of R_1 and R_2. An *unconditional join* is a join in which the restrictions on either relation are not considered. □

Join selectivity is the same as the ratio of the number of tuples participating in the unconditional join to the total number of tuples in the relation (cardinality of the relation). Join selectivity is generally different in R_1 and R_2 with respect to a join path as shown in the following example:

Example 1: Let us assume that the two relations in Figure E-4 have an 1-to-N partial-dependency relationship. Partial dependency means that every tuple in the relation R_2 that is on the N-side of the relationship has a corresponding tuple in R_1, but not vice versa [ELM 80]. Let us assume that 50% of the countries have at least one ship so that the tuples representing those countries participate in the unconditional join. Every tuple in the SHIPS relation (R_2) participates in the unconditional join according to the partial dependency. The join selectivity of the COUNTRIES relation is then 0.5, while that of the SHIPS relation is 1.0. □

R_1	COUNTRIES(Countryname, Population)
R_2	SHIPS(ShipId, Country, Crewsize, Deadweight)

Figure E-4: COUNTRIES and SHIPS relations

Definition 2: The *coupling effect (partial coupling effect)* from relation R_1 to relation R_2, with respect to each transaction, is the ratio of the number of distinct join column values of the tuples of R_1, selected according to the restriction predicate (index-processible predicate) for R_1, to the total number of distinct join column values in R_1. □

If we assume that the join column values are randomly selected, the coupling effect (partial coupling effect) from R_1 to R_2 is the same as the ratio of the number of distinct join column values of R_2 selected by the effect of the restriction predicate (index-processible predicate) for R_1 to the total number of distinct join column values in R_2 participating in the unconditional join.

Definition 3: A *coupling factor* Cf_{12} (*partial coupling factor* PCf_{12}) from relation R_1 to relation R_2 with respect to a transaction is the ratio of the number of distinct join column values of R_2, selected by both the coupling effect (partial coupling effect) frǫm R_1 (through the restriction predicate for R_1) and the join selectivity of R_2, to the total number of distinct join column values in R_2. □

According to the definition, a coupling factor can be obtained by multiplying the coupling effect (partial coupling effect) from R_1 to R_2 by the join selectivity of R_2.

Definition 4: A *partial-join cost* is the part of the join cost that represents the accessing of only one relation as well as the auxiliary structures defined for that relation. □

Definition 5: A *partial-join algorithm* is a conceptual division of the algorithm of a join method whose processing cost is a partial-join cost. □

Definition 6: The *restricted set* of relation R with respect to a transaction is the set of tuples of R selected according to the restriction predicate for R. □

Definition 7: The *partially restricted set* of relation R with respect to a transaction is the set of tuples of R selected according to the index-processible predicate for R. □

Definition 8: The *coupled set* of relation R_1 with respect to a transaction is the set of tuples in R_1 selected according to the coupling factor from R_2. □

Definition 9: The *partially coupled set* of relation R_1 with respect to a transaction is the set of tuples of R_1 selected according to the partial coupling factor from R_2. □

Definition 10: The *result set* of relation R with respect to a transaction is the intersection of the restricted set and the coupled set. Thus, the tuples in the result set satisfy all the predicates in a transaction. □

Definition 6 to Definition 10 define various subsets of the relation according to the predicates they satisfy. In Figure E-5 these subsets are graphically illustrated. Cardinalities of subsets of relation R_1 can be obtained as follows:

$$\begin{aligned}
|\text{restricted set}| &= n_{R_1} \times \text{Selectivity of the restriction predicate} \\
|\text{partially restricted set}| &= n_{R_1} \times \text{Selectivity of the index-processible predicate} \\
|\text{coupled set}| &= n_{R_1} \times Cf_{21} \\
|\text{partially coupled set}| &= n_{R_1} \times PCf_{21} \\
|\text{result set}| &= n_{R_1} \times Cf_{21} \times \text{Selectivity of the restriction predicate}
\end{aligned}$$

To estimate the selectivities of the predicates, we use the following simple scheme. If a predicate is the conjunction of simple predicates that involve single columns, its selectivity can be obtained by multiplying the selectivities of those simple predicates. The selectivity of a simple equality predicate is estimated as the inverse of the number of distinct values in the related column (column cardinality). For a simple range predicate, which involves operators such as $<$, $\leq$, $>$, $\geq$, the selectivity is arbitrarily estimated as 1/4. (A more elaborate interpolation scheme can be employed if the highest and the lowest values in the column are known.) Estimating the selectivities of more general predicates has been studied in [DEM 80].

We now introduce a function that estimates the number of block accesses when randomly selected tuples are retrieved in TID order. Various formulas have been proposed for this function [CAR 75] [ROT 74] [SEV 72] [SIL 76] [WAT 72] [WAT 75] [WAT 76] [YAO-b 77] [YUE 75]. In particular, Yao [YAO-b 77] presented the following theorem:

Theorem 1: **[YAO]** Let n records be grouped into m blocks ($1 \leq m \leq n$), each containing $p = n/m$ records. If k records are randomly selected from the n records, the expected number of blocks hit (blocks with at least one record selected) is given by

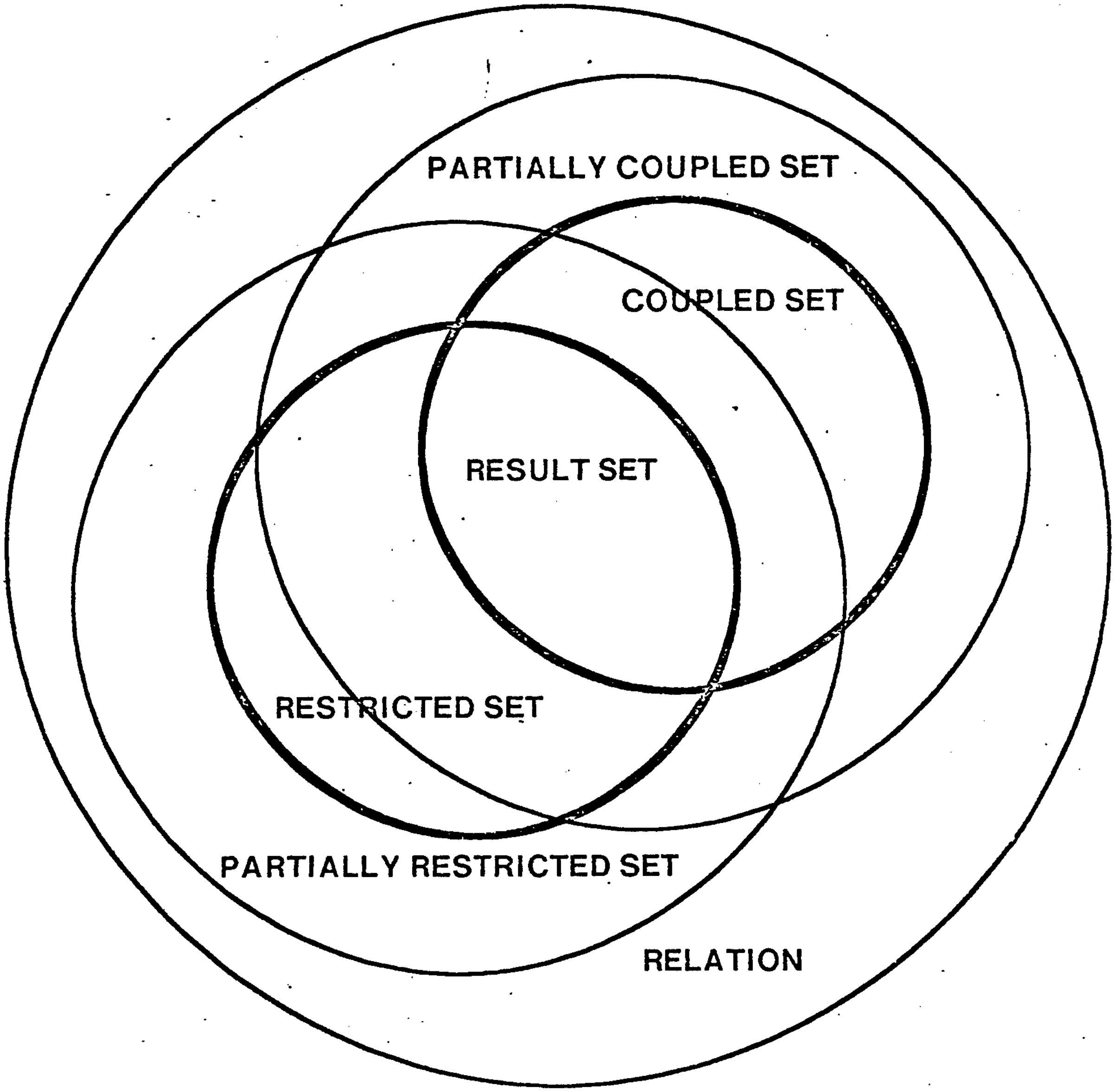


Figure E-5: Various Subsets of a Relation.

$$\begin{aligned} b(m,p,k) &= m\,[1 - \tbinom{n-p}{k}/\tbinom{n}{k}] \\ &= m\,[1 - ((n-p)!(n-k)!)/((n-p-k)!n!)] \\ &= m\,[1 - \Pi_{i=1}^{k}(n-p-i+1)/(n-i+1)] \\ & \qquad\qquad \text{when } k \leq n-p, \text{ and} \\ b(m,p,k) &= m \qquad\qquad \text{when } k > n-p. \end{aligned} \tag{E.1}$$

The function is approximately linear on k when k<<n and approaches p as k becomes large. Variations of this function and approximation formulas for faster evaluation are summarized in [WHA-a 82]. Let us note that the function is invalid if m<1.

E.5 Elementary Cost Formulas

To formulate the transaction-processing costs, we first develop cost formulas for elementary operations. Elementary cost formulas mainly concerns the costs related to a single relation and its auxiliary access structures.

When more than one tuple (or index entry) is retrieved or updated, the relative order of accessing those tuples (or index entries) becomes important in determining the cost. Below, we define four types of ordering:

- **TID order:** Tuples (or index entries) are accessed according to the order of TID. TID order can be achieved when a relation or an index is scanned or when tuples are accessed with matching keys through one or more indexes. Let us note that the index entries having the same key value are ordered by TID.

- **Random order:** Tuples (or index entries) are randomly accessed without any specific order.

- **Clustering column order:** Tuples are accessed by scanning the clustering index. This ordering specifies the orders of accessing both data tuples and index entries: both are accessed in TID order. This ordering differs from TID order of accessing tuples in that, when a tuple is accessed, the location of the corresponding entry in the clustering index is already known. We define both TID order and clustering column order as *physical order.*

- **Ordering column order:** Tuples are accessed by scanning the index of the ordering column. This ordering specifies the orders of accessing both data tuples and index entries: tuples are accessed in random order; index entries in TID order. As in clustering column order, when a tuple is accessed, the location of its corresponding entry in the ordering index is already known. This ordering occurs when the join index method is used to resolve the join predicate; here, the join column becomes the ordering column.

Elementary cost formulas are now introduced in the form of functions in the following. Each function will be followed by subsequent explanation on how it has been derived. In calculating the cost of a query, we do not include the cost of writing the result since that cost is common to all alternative processing methods and is irrelevant for optimization purposes.

- **function IA(C,R,mode):** Index Access Cost—cost for accessing the index tree starting from the root.

A. mode = Query mode

$$IA = \lceil \log_{L_C} n_R \rceil + \lceil F_C \times n_R / L_C \rceil \qquad (E.2)$$

B. mode = Insertion mode

$$IA = \lceil \log_{L_C} n_R \rceil + 1$$

C. mode = Update mode

$$IA = \lceil \log_{L_C} n_R \rceil + \lceil 0.5 \times F_C \times n_R / L_C \rceil$$

The function IA has three modes depending on the purpose of accessing the index. In query mode all the index entries having the same key value are retrieved. The first term in Equation (E.2) is the height of the index tree, and the second the number of leaf-level index blocks accessed. In insertion mode, an index entry corresponding to the inserted tuple is placed after the last entry having the same key value; thus, only one leaf-level block will be accessed. In update mode, the index entries containing the old value have to be searched to find the one having the TID of the updated tuple; thus, on the average, about half of those index entries will be searched.

- **function IS(C,R):** Index Scan Cost—cost for serially scanning the leaf-level blocks of an entire index.

$$IS = \lceil n_R / L_C \rceil$$

- **function Sort(NB,z):** Sorting Cost – cost for sorting a relation, or a part thereof, according to the values of the columns of interest.

$$SORT = 2 \times \lceil NB \rceil + 2 \times \lceil NB \rceil \times \lceil \log_z \lceil NB \rceil \rceil$$

Function Sort represents the cost of an external sort using the z-way sort merge [KNU-b 73]. NB is the number of blocks in the temporary relation containing the subtuples to be sorted after restriction and projection have been resolved. It will be noted that function Sort does not include the initial scanning cost to bring in the original relation, while it does include the cost to scan the temporary relation for the actual join after sorting (see [BLA 76]).

- **function Single-Query(R,t,mode):** Single-Relation Querying Cost – cost for retrieving tuples that satisfy the restriction predicates from a single relation.

A. If no restriction index is clustering

$$\text{Single-Query} = b(m_R, p_R, |\text{partially restricted set}|) + \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,R,\text{query mode}) \qquad (E.3)$$

B. If any restriction index is clustering

a. when $F_{cc} \times m_R \geq 1$

$$\text{Single-Query} = b(F_{cc} \times m_R, p_R, |\text{partially restricted set}|) + \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,R,\text{query mode}) \qquad (E.4)$$

b. when $F_{cc} \times m_R < 1$

$$\text{Single-Query} = F_{cc} \times b(1/F_{cc}, F_{cc} \times n_R, |\text{partially restricted set}|/F_{cc}) + \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,R,\text{query mode}) \qquad (E.5)$$

The function Single-Query has two modes: "join column included" and "join column not included". In the former mode- the join predicate is treated as another restriction predicate. Accordingly, the join column becomes a restriction column, and the join index becomes a restriction index. The partially restricted set also has to be modified. This mode is useful when considering the cost of the inner/outer-loop join method. In this method, a value is substituted for the join attribute of Other(R). Resolving the join predicate for relation R then becomes a simple restriction.

Single-relation queries are processed as follows: Each restriction index is accessed in query mode to obtain the list of TIDs satisfying the corresponding simple restriction predicate. The intersection of these TID lists is formed subsequently to locate tuples satisfying the index-processible predicate. The first terms in Equations (E.3), (E.4), and (E.5) represent the cost of accessing data tuples; the second the cost of accessing indexes.

We have two cases in calculating the cost of accessing data tuples. If no restriction index is clustering, the tuples in the partially restricted set will be spread all over m_R blocks. Since they are accessed in TID order, we obtain the first term of Equation (E.3). On the other hand, if one of the restriction indexes is clustering, the tuples to be retrieved are confined in $F_{cc} \times m_R$ blocks (let us call this a *selected area*). Since tuples are accessed in TID order within the selected area, if $F_{cc} \times m_R \geq 1$, we obtain the first term of Equation (E.4). If $F_{cc} \times m_R < 1$, however, the "b" function becomes invalid, and we need an alternative derivation. Let us assume that the selected area resides within a physical block (i.e., we ignore the case in which the selected area resides on the border of two blocks) and imagine that the file is divided into $(1/F_{cc})$ logical blocks of the same size as the selected area. Then, the probability that the selected area will be hit when all the restriction predicates except for the one matching the clustering index are applied can be obtained as the first term of Equation (E.5). It is also the probability that the physical block containing the selected area will be hit. Since this physical block is the only one that can be possibly be accessed, the number of physical blocks to be hit is equivalent to this probability.

- **function Sort-Merge(R,t):** Partial Sort-Merge Join Cost-cost for joining the relation R with another using the sort-merge join method(partial).

Sort-Merge = Single-Query (R,t,join column not included) + Sort(NB),

where

$$NB = (|\text{restricted set}|/n_R) \times H_{t,R} \times m_R\,.$$

First, tuples in the partially restricted set are retrieved using the restriction indexes (this corresponds to a single-relation query). Those tuples are sorted and stored in a temporary relation after the residual predicate and the projection are resolved; the temporary relation is subsequently read in for an actual join. The term Sort(NB) includes all the cost for this operation.

- **function Join-Index(R,t,Cf):** Partial Join-Index Join Cost-cost for joining relation R with another using the join index method(partial).

Join-Index = Index Read Cost + Data Read Cost

(Index Read Cost)

$$\text{Index Read Cost} = \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,R,\text{query mode}) + IS(\text{join index, } R)$$

(Data Read Cost)

A. If the join index is nonclustering

$$\text{Data Read Cost} = Cf \times |\text{partially restricted set}| \quad \text{(E.6)}$$

B. If the join index is clustering

$$\text{Data Read Cost} = b(m_R, P_R, Cf \times |\text{partially restricted set}|)$$

Here, the parameter Cf can be either the coupling factor or the partial coupling factor from relation Other(R) to relation R. If the tuples of R are accessed first during the join operation, Cf is a partial coupling factor since only the index-processible predicate for Other(R) can be resolved before tuples of R are accessed. On the other hand, if the tuples of Other(R) are accessed first, Cf must be a coupling factor since full restriction predicate is resolved for Other(R) beforehand. In either case, Cf can be treated as yet another restriction factor as far as relation R is concerned. It will be noted that, if all restriction columns have indexes, the partial coupling factor is equivalent to the coupling factor.

The Partial Join-Index Join Cost consists of two parts: Index Read Cost and Data Read Cost. The cost of reading relevant indexes (Index Read Cost) includes the cost of accessing all restriction indexes and the cost of scanning the join index. The cost of retrieving tuples from the relation (Data Read Cost) differs according to whether the join index is clustering or not. If the join index is not clustering, tuples are retrieved in random order as the join index is scanned. Since one block access is necessary for each tuple, we obtain Equation (E.6). If the join index is clustering, since tuples are retrieved in TID order, the "b" function has to be employed.

- **function Inner/Outer(R,t,To-or-From):** Partial Inner/Outer-Loop Join Cost—cost for joining relation R with another using the inner/outer-loop join method(partial).

A. To-or-From = From

$$\text{Inner/Outer} = \text{Single-Query}(\text{R,t,join column not included})$$

B. To-or-From = To

$$\begin{aligned}\text{Inner/Outer} = {} & |\text{restricted set of Other(R)}| \times \text{Jsel}_{\text{Other(R)}} \times \\ & \quad \text{Single-Query}(\text{R,t,join column included}) \\ & + |\text{restricted set of Other(R)}| \times (1-\text{Jsel}_{\text{Other(R)}}) \times \\ & \quad \sum_{C \in \{\text{all restriction columns and the join column having indexes}\}} \text{IA}(\text{C,R,query mode})\end{aligned}$$

The cost of the inner/outer-loop join method differs depending on the join direction, which is determined by the parameter To-or-From. If the join direction is from R to Other(R) (To-or-From = From), the processing cost for R simply becomes that of a single-relation query with the mode "join index not included". However, if the join direction is reversed (To-or-From = To), the cost for R can be obtained, in principle, by summing up the costs of single-relation queries each of which is associated with a tuple in the restricted set of Other(R). The cost formula consists of two terms.

The first term is multiplied by the function Single-Query with the mode "join column included". But, the second term is multiplied by the access costs of indexes only, because the single-relation queries corresponding to the tuples of Other(R) having join column values nonexistent in R will retrieve no data tuples.

- **function Delete(R,t,order,Ntuples-deleted)**: Deletion Cost−cost for deleting Ntuples-deleted tuples from relation R according to the order specified in parameter "order".

Delete = Data Write Cost + Index Read/Write Cost

A. Deletion is performed in physical order

(Mfactor = 1 if deletion is performed in clustering column order.)
(Mfactor = 2 if deletion is performed in TID order.)

A.1. If no restriction column is clustering

(Data Write Cost)

$$\text{Data Write Cost} = b(m_R, p_R, \text{Ntuples-deleted}) \tag{E.7}$$

(Index Read/Write Cost)

a. If the clustering column does not have an index or there is no clustering column

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes}\}} [IA(C,R,\text{update mode})+1]$$

b. If the clustering column has an index

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the clustering column}\}} [IA(C,R,\text{update mode})+1]$$
$$+ \text{Mfactor} \times b(im_{cc}, L_{cc}, \text{Ntuples-deleted})$$

A.2. If a restriction column is clustering

(Data Write Cost)

a. when $F_{cc} \times m_R \geq 1$

$$\text{Data Write Cost} = b(F_{cc} \times m_R, p_R, \text{Ntuples-deleted}) \tag{E.8}$$

b. when $F_{cc} \times m_R < 1$

$$\text{Data Write Cost} = F_{cc} \times b(1/F_{cc}, F_{cc} \times n_R, \text{Ntuples-deleted}/F_{cc}) \tag{E.9}$$

(Index Read/Write Cost)

a. If the clustering column does not have an index

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes}\}} [IA(C,R,\text{update mode}) + 1]$$

b. If the clustering column has an index

b.1. when $F_{cc} \times im_{cc} \geq 1$

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the clustering column}\}} [IA(C,R,\text{update mode}) + 1] + \text{Mfactor} \times b(F_{cc} \times im_{cc}, L_{cc}, \text{Ntuples-deleted})$$

b.2. when $F_{cc} \times im_{cc} < 1$

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the clustering column}\}} [IA(C,R,\text{update mode}) + 1] + \text{Mfactor} \times F_{cc} \times b(1/F_{cc}, F_{cc} \times n_R, \text{Ntuples-deleted}/F_{cc})$$

B. Deletion is performed in ordering column order.

(Data Write Cost)

$$\text{Data Write Cost} = \text{Ntuples-deleted} \tag{E.10}$$

(Index Read/Write Cost)

B.1. If the ordering column is not a restriction column

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the ordering column}\}} [IA(C,R,\text{update mode}) + 1]$$

B.2. If the ordering column is a restriction column

a. when $F_{\text{order col}} \times im_{\text{order col}} \geq 1$

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the ordering column}\}} [IA(C,R,\text{update mode})+1]$$
$$+\ b(F_{\text{order col}} \times im_{\text{order col}}, L_{\text{order col}}, \text{Ntuples-deleted})$$

b. when $F_{\text{order col}} \times im_{\text{order col}} < 1$

$$\text{Index Read/Write Cost} = \text{Ntuples-deleted} \times \sum_{C \in \{\text{all columns having indexes except for the ordering column}\}} [IA(C,R,\text{update mode})+1]$$
$$+\ F_{\text{order col}} \times b(1/F_{\text{order col}}, F_{\text{order col}} \times n_R, \text{Ntuples-deleted}/F_{\text{order col}})$$

The deletion cost consists of two parts: Data Write Cost and Index Read/Write Cost. The former is the cost of writing the modified data blocks out to the disk. (In formulating the deletion cost, the data blocks are assumed to have already been read in the main memory.) For each tuple deleted, the corresponding index entry should also be deleted. Thus, the latter cost includes the cost of reading in the index blocks to be modified and the cost of writing them back to the disk.

The Data Write Cost differs according to the order of deleting the tuples. If deletion is performed in physical order, the "b" function has to be employed in all cases; we have two subcases. If no restriction column is clustering, tuples can be deleted from any one of the m_R blocks; thus, we obtain Equation (E.8). On the other hand, if any restriction column is clustering, tuples to be deleted are confined in $F_{cc} \times m_R$ blocks. Hence, if $F_{cc} \times m_R \geq 1$, we obtain Equation (E.8). If $F_{cc} \times m_R < 1$, according to the same argument as has been used for the function Single-Query, we obtain Equation (E.9). If deletion is performed in ordering column order, tuples to be deleted are accessed in random order. Thus, as many block accesses are incurred as the number of tuples deleted (Equation (F.10)).

The Index Read/Write Cost is obtained as follows. In general, for each index, locating the index entry corresponding to the deleted tuple requires accessing the index from the root with update mode; writing the modified index block needs one block access. When the tuples are deleted in physical order, however, special consideration must be given to the clustering index. First, since the entries in the clustering index are deleted in TID order, the "b" function is employed. If no restriction column is clustering, the index entries to be selected are spread all over im_{cc} blocks; however, if a restriction column is clustering, those index entries are confined in $F_{cc} \times im_{cc}$ blocks, and again a consideration similar to the one applied to function Single-Query has to be made. Mfactor (multiplying factor) is 1 if only writing cost of the modified blocks of the clustering index is needed (when tuples are deleted in clustering column order). Mfactor is 2 if both reading and writing costs of index blocks are considered (when tuples are deleted in TID order). When deletion is performed according to the ordering column order, the Index Read/Write Cost is obtained just as in the case of the clustering column order, except that the ordering column replaces the clustering column.

- **function Insert(R,t,Ntuples-inserted):** Insertion Cost—cost for inserting Ntuples-inserted tuples in relation R.

Insert = Data Read Cost + Data Write Cost + Index Read/Write Cost

A. If the clustering column does not exist

Data Read Cost = Ntuples-inserted

Data Write Cost = Ntuples-inserted

$$\text{Index Read/Write Cost} = \text{Ntuples-inserted} \times \sum_{C \in \{\text{all columns having indexes}\}} [\text{IA(C,R,insertion mode)} + 1]$$

B. If the clustering column exists and has an index

Data Read Cost = Ntuples-inserted×[IA(clustering column,R,insertion mode)+1]

Data Write Cost = Ntuples-inserted

$$\text{Index Read/Write Cost} = \text{Ntuples-inserted} \times \left\{ \sum_{C \in \{\text{all columns having indexes except for the clustering column}\}} [IA(C,R,\text{update mode})+1] + 1 \right\}$$

C. If the clustering column exists, but does not have an index

$$\text{Data Read Cost} = \text{Ntuples-inserted} \times \lceil m_R/2 \rceil$$

$$\text{Data Write Cost} = \text{Ntuples-inserted}$$

$$\text{Index Read/Write Cost} = \text{Ntuples-inserted} \times \sum_{C \in \{\text{all columns having indexes}\}} [IA(C,R,\text{update mode})+1]$$

For simplicity, we consider only the cases in which tuples are inserted in random order. (Unlike the deletion cost, the insertion cost includes the cost of reading in the data blocks.) The cases in which tuples are inserted in physical order or ordering column order can be analyzed using the same technique as has been used for the deletion cost. The insertion cost consists of three parts: Data Read Cost, Data Write Cost, and Index Read/Write Cost. The first is the cost of locating the places to insert new tuples. The second is that of writing modified data blocks. The third is that of updating the indexes accordingly.

If there is no clustering column (Case A), tuples can be inserted at the end of the relation. Thus, reading and writing the block into which a tuple is to be inserted cause one block access, respectively. The location into which the index entry corresponding to the inserted tuple is to be placed can be found by accessing the index from the root with insertion mode using the value of the corresponding column of the inserted tuple as the key; this operation causes IA(C,R,insertion mode) block accesses. Function IA is invoked in insertion mode because the new index entry must have the largest TID value. Writing the modified index block causes one block access.

If the clustering column exists and has an index (Case B), the place into which the tuple is to be inserted can be found through the clustering index using insertion mode; this operation causes IA(clustering column,R,insertion mode) block accesses. One more block access is needed to read the data block. Writing the modified data block also causes one block access. The Index Read/Write Cost is obtained in a way similar to Case A, but an update mode must be used for function IA since index entries having the same key value must be ordered according to their TIDs. Excluded from the Index Read/Write Cost is the cost for reading the clustering index since it has already been included in the Data Write Cost; but, one block access must be added to account for the cost of writing the modified clustering index block.

If the clustering column exists, but does not have an index (Case C), the relation has to be sequentially searched to locate the place for insertion, causing on the average $\lceil m_R/2 \rceil$ block accesses. Writing the modified block requires one block access. As in Case B, in calculating the Index Read/Write Cost, update mode must be used for function IA.

- **function Update(R,t,order,Ntuples-updated):** Update Cost—cost for updating Ntuples-updated tuples of relation R according to the order specified in parameter "order".

A. If the clustering column is updated
Update = Delete(R,t,order,Ntuples-updated) + Insert(R,t,Ntuples-updated)

B. If the clustering column is not updated
Update = Data Write Cost + Index Read/Write Cost

B.1. Updates are performed in physical order.

(Data Write Cost)
a. If no restriction column is clustering

$$\text{Data Write Cost} = b(m_R, p_R, \text{Ntuples-updated})$$

b. If a restriction column is clustering b.1. when $F_{cc}\times m_R\geq 1$

$$\text{Data Write Cost} = b(F_{cc}\times m_R, p_R, \text{Ntuples-updated})$$

b.2. when $F_{cc}\times m_R<1$

$$\text{Data Write Cost} = F_{cc}\times b(1/F_{cc}, F_{cc}\times n_R, \text{Ntuples-updated}/F_{cc})$$

(Index Read/Write Cost)

$$\text{Index Read/Write Cost} = \text{Ntuples-updated}\times 2\times \{\sum_{C\in\{\text{all updated columns having indexes}\}} [IA(C,R,\text{update mode})+1]$$

B.2. Updates are performed in ordering column order

(Data Write Cost)

$$\text{Data Write Cost} = \text{Ntuples-updated}$$

(Index Read/Write Cost)

a. If the ordering column does not have an index or is not updated

$$\text{Index Read/Write Cost} = \text{Ntuples-updated}\times 2\times \{\sum_{C\in\{\text{all updated columns having indexes}\}} [IA(C,R,\text{update mode})+1]$$

b. If the ordering column has an index and is updated

$$\text{Index Read/Write Cost} = \text{Ntuples-updated}\times 2\times$$

$$\{\sum_{C\in\{\text{all updated columns having indexes except for the ordering column}\}} [IA(C,R,\text{update mode})+1]$$

$$+\,b(im_{\text{order col}}, L_{\text{order col}}, \text{Ntuples-updated})$$

$$+\,\text{Ntuples-updated}\times[IA(\text{ordering column},R,\text{update mode})+1]$$

First, let us consider the case in which the clustering column is updated (Case A). In this case an update operation can be considered as a deletion followed by an insertion. Deletion is performed according to the order specified for update, but insertion follows a random order since the column is updated to arbitrary values specified in the transaction. Although it is conceivable that the order could be preserved after the update (e.g., new value = old value + 10), we ignore this case since detecting this property requires understanding of the semantics of the transaction which is difficult to achieve at the optimizer level.

Next, we consider the case in which the clustering column is not updated. The update cost consists of the Data Write Cost and the Index Read/Write Cost. The Data Write Cost is identical to that of the deletion cost. The Index Read/Write Cost consists of two parts: cost of deleting index entries for old values and the cost of inserting index entries for new values. In general, locating the index entry for the old value requires accessing the index from the root in update mode; finding the location where the new value is to be placed also requires accessing the index in update mode since index entries having the same key values are ordered according to their TIDs. Thus, we have a factor of 2. Writing the modified index block causes one block access. Index accessing cost for an ordering column needs special attention. Since tuples are accessed in ordering column order, the index must have already been read, and the "b" function should be used for the cost of writing the modified index blocks. The cost of inserting index entries for new values are identical to those of other indexes.

One problem is worth note when updates are performed in the following situations:

- The clustering column is updated while tuples are located by a relation scan.
- The clustering column is updated while tuples are located in clustering column order.
- The ordering column is updated while tuples are located in ordering column order.

In these situations the problem is that an updated tuple can be encountered more than once since the position of the tuple (or index entry) moves after its update [SCH 81] [STO 76]. Two solutions are suggested to avoid this anomaly. One adopted in [STO 76] is the *deferred update*. Here, updated tuples (or index entries) are stored in a temporary file and merged to the main file (or index) after update has been completed. Another strategy suggested in [SCH 81] is to avoid the above three situations by choosing an alternative access path in processing transactions. Although we included cost formulas for all cases for simplicity, if desired, the exceptional cases can always be avoided and the corresponding cost formulas ignored.

E.6 Cost Formulas for Processing Transactions

The costs for processing transactions are derived using the elementary cost formulas defined in Section E.5. Transactions are classified into the following eight types:

SQ:	Single-relation (one-variable) queries
SD:	Single-relation deletion transactions.
SU:	Single-relation update transactions.
INS:	Insertion transactions (single-relation transactions only).
AQ:	Single-relation queries having aggregate operators in their SELECT clauses, or GROUP BY constructs [CHA 76] or both.
JQ:	Two-relation (two-variable) queries having join predicates (i.e., two-relation joins)
JU:	Update transactions having join predicates.
JD:	Deletion transactions having join predicates.

We introduce below the cost formulas for each type of transaction. For transactions containing join predicates, costs are calculated for all combinations of partial-join algorithms. The combination is specified in the parenthesis: the first entry represents the partial-join algorithm for R_1, and the second for R_2. The join direction is also specified, when relevant, by an arrow from the starting relation to the other relation. The factor "freq" stands for the relative frequency of occurrence of a transaction.

1. SQ

Cost = freq * Single-Query(R,t,join column not included)

2. SD

Cost = freq * [Single-Query(R,t,join column not included)+
Delete(R,t,TID order,|restricted set|)]

3. SU

Cost = freq * [Single-Query(R,t,join column not included)+
Update(R,t,TID order,|restricted set|)]

4. INS

Cost = freq * Insert(R,t,random order, number of tuples inserted)

5. AQ

(Sort-Merge Method, –)

Cost = freq * Sort-Merge(R,t)

(Join Index Method, –)

Cost = freq * Join-Index(R,t)

6. JQ

(Sort-Merge Method, Sort-Merge Method)

Cost1 = freq * [Sort-Merge(R_1,t) + Sort-Merge(R_2,t)]

(Sort-Merge Method, Join Index Method)

Cost2 = freq * [Sort-Merge(R_1,t) + Join-Index(R_2,t,Cf_{12})]

(Join Index Method, Sort-Merge Method)

Cost3 = freq * [Join-Index(R_1,t,Cf_{21}) + Sort-Merge(R_2,t)]

(Join Index Method, Join Index Method)

a. $R_1 \rightarrow R_2$ (tuples in R_1 are accessed first)

Cost4 = freq * [Join-Index(R_1,t,PCf_{12})+Join-Index(R_2,t,Cf_{12})]

b. $R_2 \rightarrow R_1$ (tuples in R_2 are accessed first)

Cost5 = freq * [Join-Index(R_1,t,Cf_{21})+Join-Index(R_2,t,PCf_{12})]

(Inner/Outer-Loop Join Method, Inner/Outer-Loop Join Method)

a. $R_1 \rightarrow R_2$

$\text{Cost6} = \text{freq} * [\text{Inner/Outer}(R_1,t,\text{From})+\text{Inner/Outer}(R_2,t,\text{To})]$

b. $R_2 \rightarrow R_1$

$\text{Cost7} = \text{freq} * [\text{Inner/Outer}(R_1,t,\text{To})+\text{Inner/Outer}(R_2,t,\text{From})]$

7. JU

(Sort-Merge Method, Sort-Merge Method) Not allowed

(Sort-Merge Method, Join Index Method) Not allowed

(Join Index Method, Sort-Merge Method)

$\text{Costl} = \text{freq} * [\text{Join-Index}(R_1,t,Cf_{21}) + \text{Sort-Merge}(R_2,t)+$
$\text{Update}(R_1,t,\text{ordering column}=\text{join column},|\text{result set of } R_1|)]$

(Join Index Method, Join Index Method)

a. $R_1 \rightarrow R_2$

$\text{Cost2} = \text{freq} * [\text{Join-Index}(R_1,t,PCf_{21}) + \text{Join-Index}(R_2,t,Cf_{12})+$
$\text{Update}(R_1,t,\text{ordering column}=\text{join column},|\text{result set of } R_1|)]$

b. $R_2 \rightarrow R_1$

$\text{Cost3} = \text{freq} * [\text{Join-Index}(R_1,t,Cf_{21}) + \text{Join-Index}(R_2,t,PCf_{12})+$
$\text{Update}(R_1,t,\text{ordering column}=\text{join column},|\text{result set of } R_1|)]$

(Inner/Outer-Loop Join Method, Inner/Outer-Loop Join Method)

a. $R_1 \rightarrow R_2$

$\text{Cost4} = \text{freq} * [\text{Inner/Outer}(R_1,t,\text{From}) + \text{Inner/Outer}(R_2,t,\text{To})$
$\text{Update}(R_1,t,\text{TID order}, |\text{result set of } R_1|)]$

b. $R_2 \rightarrow R_1$ Not allowed

8. JD

The cost formulas are identical to those of type JU transactions except that function Delete replaces function Update.

Cost formulas for type SQ, SD, SU, and INS transactions are directly derived from the definitions of elementary cost formulas.

A type AQ transaction is essentially a partial-join between the GROUP BY column and the relation itself as far as the I/O access cost is concerned. Thus, if the sort-merge method(partial) is used, the relation is sorted according to the join column order so that tuples in the same group are clustered together. The sorted temporary relation is subsequently scanned to process the transaction. The join index method(partial) can also be used since, by scanning the join index, tuples in the same group can be retrieved consecutively. The inner/outer-loop join method(partial) is not applicable in processing a type AQ transaction.

The cost of a type JQ transaction is composed of two partial-join costs: one for relation R_1, the other for relation R_2. Except for the cases in which the sort-merge method(partial) is included the cost differs depending on the direction of the join, i.e., depending on which relation's tuples are to be accessed first. Specifically, if the join index method(partial) is used for both relations, the partial coupling factor must be used for the relation to be accessed first; the coupling factor for the other. Also, if the inner/outer-loop join method is used, the direction must be specified explicitly as a parameter in function Inner/Outer.

The cost of a type JU or JD transaction also consists of two partial-join costs and, in addition, update or deletion cost. Most of the join methods described in Section E.3 can be applied to a type JU or JD transaction as well, with some exceptions: the sort-merge method(partial) cannot be used for the relation to be updated (R_1); the inner/outer-loop join method is not allowed when the join is directed towards relation R_1.

E.7 Summary and Conclusion

A comprehensive set of formulas for estimating transaction-processing costs in relational database systems has been developed. First, terminology has been defined in Section E.4 to provide a mechanism for understanding interaction among relations in multiple-file environment. Next, a set of elementary cost formulas has been developed for elementary access operations. In doing that, four types of orderings have been defined to characterize the order of accessing tuples. Finally, transactions have been classified into eight types, and the cost formulas for each type have been derived as composites of elementary cost formulas.

The cost formulas have been fully implemented in the Physical Database Design Optimizer (PhyDDO) – an experimental system for developing various heuristics for the multiple-file physical database design described in Appendix section K. The system accepts the eight types of transactions described in Section E.6 and produces the optimal configuration of the physical database.

The formulas developed in this chapter use a higher level abstraction compared with other cost models that incorporate more details of the storage structure [WIE 83] [SEN 69]. In particular, the cost model we used in this chapter uniformly account for the number of block accesses without differentiating sequential block accesses and random block accesses. This assumption is valid in DBMS's that do not explicitly exploit sequential storage allocation.

Our model also uses a very simple assumption on the buffer strategy. (It has been assumed that a new block access is needed unless two data elements consecutively accessed reside in the same block.) Although the validity of this assumption has not been validated with actual databases in this paper, we believe that it will be sufficient for most practical cases. Experiments based on simulation using the PhyDDO further supports that claim.

The main contribution of this paper is to present a coherent and complete set of cost formulas for various types of transactions including queries, update, deletion, and insertion transactions. We believe that the techniques employed in this paper will provide a useful tool for future research on developing cost formulas for various database systems.

Appendix F. Index Selection in Relational Databases

This paper has been submitted for publication. For convenience all references have been moved to the end of the thesis.

Index Selection in Relational Databases

by

Kyu-Young Whang
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

An index selection algorithm for relational databases is presented. The problem concerns finding an optimal set of indexes that minimizes the average transaction-processing cost. This cost is measured in terms of the number of I/O accesses. The algorithm presented employes a heuristic approach called *DROP heuristic*. In an extensive test performed to determine the optimality of the algorithm, the algorithm found optimal solutions in all cases. The time complexity of the algorithm shows a substantial improvement when compared with the approach of exhaustively searching through all possible alternatives. This algorithm is further extended to incorporate the clustering property (the relation is stored in a sorted order) and also is extended for application to multiple-file databases.[5]

F.1 Introduction

We consider the problem of selecting a set of indexes that minimizes the transaction-processing cost in relational databases. The cost of a transaction is measured in terms of the number of I/O accesses.

The index selection problem has been studied extensively by many researchers. A pioneering work based on a simple cost model appeared in [LUM 71]. A more detailed cost model incorporating index storage cost as well as retrieval and index maintenance cost was developed in [AND 77]. Some approaches [KIN 74], [STO 74] attempted to formalize the problem to obtain analytic results in some restricted cases. In a more theoretical approach Comer [COM 78] proved that a simplified version of the index selection problem is NP-complete. Thus, the best known

[5] This work was supported by the Defense Advanced Research Project Agency under the KBMS Project, Contract N39-82-C-0250.

Author's current address: Computer Systems Laboratory ERL 416, Stanford University, Stanford, California 94305

algorithm to find an optimal solution would have an exponential time complexity. In an effort to find a more efficient algorithm, Schkolnick [SCH 75] discovered that, if the cost function satisfies a property called *regularity*, the complexity of the optimal index selection algorithm can be reduced to less than exponential. Hammer and Chan [HAM 76] took a somewhat different approach and developed a heuristic algorithm that drastically reduced the time complexity. However, the optimality of this algorithm has not been investigated.

Although there has been considerable efforts on developing algorithms for index selection, most past research has concentrated on single-file cases. Furthermore, incorporation of the primary structure (the clustering property) of the file has remained to be solved. The purpose of this paper is to develop an index selection algorithm with a reasonable efficiency that can be extended to multiple-file environments as well as extended to incorporate the clustering property.

The approach presented in this paper bears some resemblance to the one introduced by Hammer and Chan [HAM 76]. But, there is one major modification: the DROP heuristic [FEL 66] is employed instead of the ADD heuristic [KUE 63]. The DROP heuristic attempts to obtain an optimal solution by incrementally dropping indexes starting from a full index set. On the other hand, the ADD heuristic adds indexes incrementally starting from an initial configuration without any index to reach an optimal solution.

Since we are pursuing a heuristic approach for index selection, the actual result is suboptimal. However, in an extensive test performed for validation, the algorithm found optimal solutions in all cases. (On the other hand, the ADD heuristic found nonoptimal solutions in several occasions.)

We present first the index selection algorithm for single-file databases without the clustering property. This algorithm is tested for its validation with 24 randomly generated input situations, and the result compared with the optimal solutions generated by exhaustively searching through all possible index sets. This algorithm is then extended to incorporate the clustering property.

Extension to multiple-file cases is subsequently considered. Section F.2 introduces major assumptions, while Section F.3 describes classes of transactions we consider and their cost functions. The index selection algorithm and its time complexity are presented in Section F.4. Discussed in Section F.5 is the result of the test performed for validation of the algorithm. The algorithm is extended to incorporate the clustering property in Section F.6 . Finally, discussed in Section F.7 is an extension of the algorithm for application to multiple-file databases.

F.2 Assumptions

We assume that the relation is stored in a secondary storage medium, which is divided into fixed-size units called blocks [WIE 83]. In processing a transaction the number of I/O accesses necessary to bring the blocks into the main memory depends on the specific buffer strategy. We assume, however, the following simple strategy: no block access will be necessary if the next tuple (or index entry) to be accessed resides in the same block as that of the current tuple (or index entry); otherwise, a new block access is necessary. We also assume that all TID (tuple identifier) manipulations can be performed in the main memory without any need for I/O accesses.

We consider only conjunctive predicates consisting of simple equality predicates (e.g., A = 'a'). The selectivities of each simple predicate is estimated as the inverse of the corresponding column cardinality. If a predicate is a conjunction of simple predicates, its selectivity is obtained by multiplying the selectivities of those simple predicates. More general predicates can be incorporated if a more elaborate scheme for estimating the selectivities [DEM 80] is employed.

We assume that a B^+-tree index [COM 79] can be defined for a column of a relation. The leaf-level of the index consists of (key, TID) pairs for every tuple in that relation and the leaf-level blocks are chained so that the index can be scanned without traversing the index tree. Entries having the same key value are ordered by TID. When index entries are inserted or deleted, we assume that splits or concatenations of index blocks are rather infrequent so that modifications are mainly done on leaf-level blocks. Let us note that this model of storage structure is not essential for the validity of the algorithm to be presented, but is necessary for implementation.

F.3 Transaction Model

We consider four types of transactions: query, update, deletion, and insertion transactions. The classes of transactions for those types are shown in Figures F-1 to F-4.

```
SELECT <list of columns>
FROM   R
WHERE  P
```

Figure F-1: General Class of Queries Considered.

```
UPDATE R
SET    R.A = <new value_A>,
SET    R.B = <new value_B>,
       .
       .
       .
WHERE  P
```

Figure F-2: General Class of Update Transactions Considered.

```
DELETE R
WHERE  P
```

Figure F-3: General Class of Deletion Transactions Considered.

```
INSERT  INTO R: <list of column values>
```

Figure F-4: General Class of Insertion Transactions Considered.

In Figures F-1 to F-4 "P" stands for the restriction predicate that selects the relevant tuples. We call the columns appearing in P *restriction columns.*

Cost formulas for those transactions are now introduced in the form of functions. Each function will be followed by subsequent explanation on how it has been derived. In calculating the cost of a query we do not include the cost of writing the result since that cost is independent of the index set and, accordingly, irrelevant for optimization purposes. We also assume that, in resolving predicates, all the available indexes are utilized even if some index might increase the processing cost due to the access cost of the index itself.

We define the following notation:

C : A column.

n : Number of tuples in the relation (cardinality).

p : Blocking factor of the relation.

L_C : Blocking factor of the index for column C.

F_C : Selectivity of column C or of its index

m : Number of blocks in the relation, which is equal to n/p.

t : A transaction.

restricted set : Set of tuples that satisfy all the restriction predicates.
Equivalent to $(\prod_{C \in \{\text{all restriction columns}\}} F_C) \times n$.

partially restricted set : Set of tuples that satisfy the restriction predicates that can be resolved through indexes.
Equivalent to $(\prod_{C \in \{\text{all restriction columns having indexes}\}} F_C) \times n$.

- **function b(m,p,k):** cost for accessing k randomly selected tuples in TID order.

$$\begin{aligned} b(m,p,k) &= m\,[1 - \tbinom{n-p}{k}/\tbinom{n}{k}] \\ &= m\,[1 - ((n-p)!(n-k)!)/((n-p-k)!n!)] \\ &= m\,[1 - \textstyle\prod_{i=1}^{k}(n-p-i+1)/(n-i+1)] \\ &\qquad\qquad \text{when } k \leq n-p \text{, and} \\ b(m,p,k) &= m \qquad\qquad \text{when } k > n-p. \end{aligned} \tag{F.1}$$

The function is approximately linear on k when k<<n and approaches p as k becomes large. Equation (F.1) is an exact formula derived by Yao [YAO-b 77]. Variations of this function and approximation formulas for faster evaluation are summarized in [WHA-a 82].

- **function IA(C,mode):** cost for accessing a B^+-tree index from the root.

A. mode = Query mode (F.2)

$$IA = \lceil \log_{L_C} n \rceil + \lceil F_C \times n / L_C \rceil$$

B. mode = Insertion mode

$$IA = \lceil \log_{L_C} n \rceil + 1$$

C. mode = Update mode

$$IA = \lceil \log_{L_C} n \rceil + \lceil 0.5 \times F_C \times n / L_C \rceil$$

The function IA has three modes depending on the purpose of accessing the index. In query mode all the index entries having the same key value are retrieved. The first term in Equation (F.2) is the height of the index tree, and the second the number of leaf-level index blocks accessed. In insertion mode an index entry corresponding to the inserted tuple is placed after the last entry having the same key value; thus, only one leaf-level block will be accessed. In update mode the index entries containing the old value have to be searched to find the one having the TID of the updated tuple; thus, on the average, about half of the index entries will be searched.

- **function Query(t):** cost for processing a query

$$\text{Query} = b(m,p,|\text{partially restricted set}|) + \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,\text{query mode}) \qquad (F.3)$$

Queries are processed as follows. Indexes of all restriction columns are accessed in query mode to obtain the sets of TIDs satisfying the corresponding simple restriction predicates. The intersection of these TID sets is formed subsequently to locate tuples in partially restricted set. These tuples are retrieved and produced as output after the remaining restriction predicates are resolved. The first term in Equation (F.3) represents the cost of accessing data tuples; the second the cost of accessing indexes.

- **function Update(t):** cost for processing an update transaction.

$$\text{Update} = \text{Query}(t) \qquad (F.4)$$

$$+ \; b(m,p,|\text{restricted set}|)$$
$$+ \; |\text{restricted set}| \times 2 \times \sum_{C \in \{\text{all updated columns having indexes}\}} [IA(C,\text{update mode}) + 1]$$

The update cost consists of three parts: the first term of Equation (F.4) represents the cost of reading in blocks containing the tuples to be deleted; the second term the cost of writing out modified blocks; and the third term the cost of updating corresponding indexes. The third term is again divided into two parts: the cost of deleting index entries for old values and that of inserting index entries for new values. Since these two parts have the same value, a factor of 2 is introduced. Let us note that, even for insertion of new index entries, update mode is specified for function IA since index entries having the same key value must be ordered according to their TIDs.

- **function Delete(t):** cost for processing a deletion transaction.

$$\text{Delete} = \text{Query}(t)$$
$$+ \; b(m,p,|\text{restricted set}|)$$
$$+ \; |\text{restricted set}| \times \sum_{C \in \{\text{all columns having indexes}\}} [IA(C, \text{update mode}) + 1]$$

The deletion cost is the same as the update cost except that the third term of the cost function represents the cost of deleting index entries for all existing indexes.

- **function Insert(t,Ntuples-inserted):** cost for processing an insertion transaction.

$$\text{Insert} = \text{Ntuples-inserted}$$
$$\times \; (1 + 1 + \sum_{C \in \{\text{all columns having indexes}\}} [IA(C,\text{insertion mode}) + 1])$$

Three parts contribute to the insertion cost: the cost of locating the place to insert a new tuple (one I/O access); the cost of writing the modified block (one I/O access); and the cost of modifying all existing indexes accordingly. In the third part function IA is called in insertion mode since the new index entry is always added at the end of the list of index entries having the same key value.

F.4 Index Selection Algorithm (DROP heuristic)

Input:

- Usage information: A set of various query, update, insertion, and deletion transactions with their relative frequencies.
- Data characteristics: Relation cardinality, blocking factor, selectivities and index blocking factors of all columns.

Output:

- The optimal (or suboptimal) index set.

Algorithm 1:

1. Start with a full index set.
2. Try to drop one index at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index that yields the maximum cost benefit when dropped.
3. Drop that index.
4. Repeat Steps 2 and 3 until there is no further reduction in the cost.
5. Try to drop two indexes at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index pair that yields the maximum cost benefit when dropped.
6. Drop that pair.
7. Repeat Steps 5 and 6 until there is no further reduction in the cost.
8. Repeat Steps 5, 6, and 7 with three indexes, four indexes, ..., up to k (k must be predefined) indexes at a time.

The variable k, the maximum number of indexes that are dropped together at a time, must be supplied to the algorithm by the user. We believe, however, that k=2 suffices in most practical cases. In fact, in all the tests performed to validate the index selection algorithms, the maximum value of k actually used was 2.

The time complexity of the algorithm is $O(g \times v^{k+1})$, where g is the number of transactions specified in the usage information, v the number of columns in the relation, and k the maximum number of columns considered together in the algorithm. The time complexity is estimated in terms of the number of calls to the cost evaluator which is the costliest operation in the design process. In the algorithm the cost evaluator is called for every k-combination of columns of the relation, and for every transaction in the usage information. This contributes the order of $g \times v^k$. The procedure is repeated until there is no further reduction in the cost. Since the number of repetitions is proportional to v, the overall time complexity is $O(g \times v^{k+1})$.

F.5 Validation of the Algorithm

An important task in developing heuristic algorithms is their validation. In this section the result of an extensive test performed to validate the index selection algorithm (DROP heuristic) will be presented. In particular, we try to measure the deviations of the heuristic solutions from the optimal ones for various input situations generated using different parameters. (These parameters were chosen from practically important ranges.) For a relation having many columns identifying the optimal solution itself is a difficult, often impossible, task. Therefore, in the tests, the number of columns in a relation is restricted to be ten. Optimal solutions are then obtained by exhaustively searching through all possible alternatives (2^{10} combinations).

The input situations are generated as follows:

1. Two sets of the relation cardinality and column cardinalities are used: in Set 1 the relation cardinality is 1000; in Set 2 it is 100,000. The column cardinalities are randomly generated between 1 and the relation cardinality with a logarithmically uniform distribution.

2. Two sets of the blocking factor and index blocking factors are used: 1) 10 and 100; 2) 100 and 1000. The index blocking factors are assumed to be identical for all indexes.

3. The usage information includes 30 transactions and their relative frequencies. Among them there are 21 queries, 4 to 5 update transactions, 3 to 4 deletion transactions, and 1 insertion transaction. Three sets of transactions are used. For each set, transactions are randomly generated as follows: for queries and deletion transactions 1 to 3 (numbers are

randomly selected) columns are randomly selected as restriction columns; for update transactions 1 to 3 columns are randomly selected as updated columns and as restriction columns.

4. Two sets of relative frequencies are used. In Set 1 all transactions initially have identical frequencies. Later, the frequencies of deletion and insertion transactions are multiplied by an adjusting factor so as to keep the number of indexes in the result between 3 and 7. This adjustment is made to avoid extreme cases in which a full index set or an empty index set is the optimal solution. For Set 2 the relative frequencies of transactions are randomly generated between 100 and 500 with an interval of 50 between adjacent values.

The scheme described above generates 24 different input situations, one of which is shown in Figure F-5. The test results for both Drop and Add heuristics are summarized in Table 1. In the first column of Table 1 the first digit of the input situation number represents the set of the relational cardinality, the second the set of the blocking factor and index blocking factors, the third the set of transactions, and the last the set of relative frequencies of transactions. The second column of the table shows the number of indexes present in the optimal solution. The CPU time shows the performance of the algorithms when run in a DEC-2060. The situations in which any deviation occurred are given percent deviations. Marked by "opt" are the situations in which an optimal solution was found.

```
!Input Situation 2132!
Schema
    Relations
        Relation            R
            Relcard         100000
            Nblocks         10000
            Blkfac          10

            Column          C1
                Colcard     409
                Niblk       1000
                Iblkfac     100

            Column          C2
                Colcard     1333
                Niblk       1000
                Iblkfac     100

            Column          C3
                Colcard     180
                Niblk       1000
                Iblkfac     100

            Column          C4
                Colcard     1
                Niblk       1000
                Iblkfac     100

            Column          C5
                Colcard     1108
                Niblk       1000
                Iblkfac     100

            Column          C6
```

```
                Colcard         678
                Niblk           1000
                Iblkfac         100

            Column              C7
                Colcard         175
                Niblk           1000
                Iblkfac         100

            Column              C8
                Colcard         64
                Niblk           1000
                Iblkfac         100

            Column              C9
                Colcard         194
                Niblk           1000
                Iblkfac         100

            Column              C10
                Colcard         328
                Niblk           1000
                Iblkfac         100

Usage
    Transaction 1
        Type    SQ      FREQ    500
        Select  R.C1
        From    R
        Where   R.C7 ="a"       AND
                R.C10="b"

    Transaction 2
        Type    SQ      FREQ    100
        Select  R.C1
        From    R
        Where   R.C6 ="a"       AND
                R.C8 ="b"       AND
                R.C9 ="c"

    Transaction 3
        Type    SQ      FREQ    200
        Select  R.C1
        From    R
        Where   R.C3 ="a"       AND
                R.C4 ="b"       AND
                R.C9 ="c"

    Transaction 4
        Type    SQ      FREQ    100
        Select  R.C1
        From    R
        Where   R.C6 ="a"

    Transaction 5
        Type    SQ      FREQ    250
        Select  R.C1
        From    R
        Where   R.C8 ="a"       AND
                R.C2 ="b"

    Transaction 6
        Type    SQ      FREQ    50
        Select  R.C1
        From    R
        Where   R.C5 ="a"       AND
                R.C9 ="b"

    Transaction 7
        Type    SQ      FREQ    450
        Select  R.C1
        From    R
        Where   R.C7 ="a"       AND
                R.C10="b"

    Transaction 8
        Type    SQ      FREQ    100
        Select  R.C1
        From    R
        Where   R.C8 ="a"

    Transaction 9
        Type    SQ      FREQ    250
        Select  R.C1
        From    R
        Where   R.C3 ="a"       AND
                R.C2 ="b"

    Transaction 10
        Type    SQ      FREQ    450
        Select  R.C1
        From    R
        Where   R.C7 ="a"       AND
```

```
            R.C3 ="b"

Transaction 11
    Type    SQ      FREQ    500
    Select  R.C1
    From    R
    Where   R.C4 ="a"       AND
            R.C7 ="b"

Transaction 12
    Type    SQ      FREQ    250
    Select  R.C1
    From    R
    Where   R.C10="a"

Transaction 13
    Type    SQ      FREQ    150
    Select  R.C1
    From    R
    Where   R.C8 ="a"       AND
            R.C6 ="b"

Transaction 14
    Type    SQ      FREQ    250
    Select  R.C1
    From    R
    Where   R.C5 ="a"       AND
            R.C2 ="b"

Transaction 15
    Type    SQ      FREQ    100
    Select  R.C1
    From    R
    Where   R.C4 ="a"

Transaction 16
    Type    SQ      FREQ    150
    Select  R.C1
    From    R
    Where   R.C4 ="a"       AND
            R.C3 ="b"

Transaction 17
    Type    SQ      FREQ    350
    Select  R.C1
    From    R
    Where   R.C1 ="a"       AND
            R.C4 ="b"       AND
            R.C3 ="c"

Transaction 18
    Type    SQ      FREQ    150
    Select  R.C1
    From    R
    Where   R.C3 ="a"       AND
            R.C9 ="b"

Transaction 19
    Type    SQ      FREQ    150
    Select  R.C1
    From    R
    Where   R.C5 ="a"

Transaction 20
    Type    SQ      FREQ    300
    Select  R.C1
    From    R
    Where   R.C8 ="a"

Transaction 21
    Type    SQ      FREQ    400
    Select  R.C1
    From    R
    Where   R.C4 ="a"       AND
            R.C5 ="b"

Transaction 22
    Type    SU      FREQ    200
    Update  R
    Set     R.C10="a"
    Where   R.C9 ="f"

Transaction 23
    Type    SU      FREQ    300
    Update  R
    Set     R.C1 ="a",
    Set     R.C10="b",
    Set     R.C8 ="c"
    Where   R.C4 ="d"

Transaction 24
    Type    SU      FREQ    300
    Update  R
```

```
        Set      R.C3 ="a",
        Set      R.C6 ="b"
        Where    R.C8 ="c"          AND
                 R.C5 ="d"

    Transaction 25
        Type     SU       FREQ      250
        Update   R
        Set      R.C2 ="a"
        Where    R.C3 ="c"

    Transaction 26
        Type     SD       FREQ      50
        Delete   R
        Where    R.C8 ="d"          AND
                 R.C6 ="e"

    Transaction 27
        Type     SD       FREQ      150
        Delete   R
        Where    R.C1 ="d"          AND
                 R.C8 ="e"          AND
                 R.C7 ="f"

    Transaction 28
        Type     SD       FREQ      250
        Delete   R
        Where    R.C6 ="c"          AND
                 R.C5 ="d"

    Transaction 29
        Type     SD       FREQ      200
        Delete   R
        Where    R.C7 ="f"          AND
                 R.C4 ="g"

    Transaction 30
        Type     INS      FREQ      150
        Insert   INTO     R:
                 <"a1","a2","a3","a4","a5","a6","a7","a8","a9","a10">
```

Figure F-5: An Example Input Situation.

In all situations tested the DROP heuristic found optimal solutions. Although the test is by no means exhaustive, the result is a good indication that the DROP heuristic will perform well in many practical situations. In comparison, the ADD heuristic produced nonoptimal solutions in six cases; the maximum deviation encountered was 21.17%. One possible reason why the ADD heuristic does not perform well is the following. In the ADD heuristic, when the first index is added, the cost changes drastically causing an abrupt change in the design process. But, in the DROP heuristic, dropping indexes causes a smooth transition in the design process since dropping one index does not make a big change in the cost due to the presence of other indexes compensating for one another.

As we can see in Table 1, an exhaustive search takes excessive computation time; in comparison, the DROP heuristic is far more efficient without significant loss of accuracy. Obviously, for larger input situations, the exhaustive-search method will become intolerably time-consuming. In these cases, heuristic algorithms such as the DROP heuristic may be the only ones applicable.

Table 1. Accuracy and Performance of the Index Selection Algorithm.

Input Situation	Number of Indexes	CPU time(seconds) / Deviation(%)				
		Algorithm 1		ADD Heuristic		Ex. Search
1111	7	2.3	opt	2.0	0.21	36
1112	6	2.2	opt	2.1	opt	36
1121	6	2.4	opt	2.0	1.23	37
1122	6	2.3	opt	2.1	1.17	37
1131	6	2.5	opt	2.1	opt	39
1132	7	2.5	opt	2.1	1.17	39
1211	5	3.1	opt	1.7	opt	32
1212	3	1.9	opt	1.6	opt	31
1221	4	2.1	opt	1.7	opt	32
1222	5	2.0	opt	1.7	opt	32
1231	4	2.3	opt	1.7	opt	35
1232	5	2.2	opt	1.7	opt	35
2111	4	2.4	opt	2.1	16.71	39
2112	5	2.5	opt	2.1	21.17	40
2121	6	2.3	opt	2.0	opt	38
2122	7	2.5	opt	2.0	opt	37
2131	6	2.6	opt	2.2	opt	40
2132	6	2.7	opt	2.2	opt	40
2211	6	2.6	opt	2.0	opt	36
2212	4	2.4	opt	1.9	opt	36
2221	6	2.4	opt	1.9	opt	34
2222	6	2.3	opt	1.9	opt	34
2231	5	2.4	opt	2.0	opt	38
2232	5	2.4	opt	2.0	opt	38

F.6 Index Selection when the Clustering Column Exists

Incorporation of the clustering property to the index selection algorithm is straightforward. Two algorithms for this extension are presented below:

Algorithm 2:

1. For each possible clustering column in the relation perform index selection.
2. Save the best configuration.

Algorithm 3:

1. Perform index selection with the clustering column determined in Step 2 of the last iteration. (During the first iteration it is assumed that there is no clustering column.)

2. Perform clustering design with the index set determined in Step 1. The clustering property is assigned to each column in turn, and the best clustering column is selected.

3. Steps 1 and 2 are iterated until the improvement in the cost through the loop is less than a predefined value (e.g., 1%).

Algorithm 2 is a pseudo enumeration since index selection is repeated for every possible clustering column position. Naturally, Algorithm 2 has a higher time complexity compared with Algorithm 3, but has a better chance of finding an optimal solution. Both algorithms have been implemented and tested as a part of Physical Database Design Optimizer (PhyDDO)—an experimental system for developing various heuristics for the multiple-file physical database design K. In most cases tested they found optimal solutions. Let us note that the cost formula have to be modified in the presence of the clustering column. A complete set of cost formulas for multiple-file relational databases with the clustering property can be found in Appendix E.

F.7 Index Selection for Multiple-File Databases

Extension of the index selection algorithm for application to multiple-file databases is also straightforward. The extended algorithm (let us call it Algorithm 4) is almost identical to Algorithm 1 except for the followings:

1. The entire database is designed at the same time. It is done by treating all columns in the database uniformly as if they were in a single relation.

2. When evaluating transactions involving more than one relation, the optimizer [SEL 79], [STO 76] has to be invoked to find the optimal sequence of access operations.

Algorithm 4 has also been implemented and successfully tested as a part of the Physical Database Design Optimizer.

F.8 Summary and Conclusion

Algorithms for the optimal index selection in relational databases have been presented. Algorithm 1, which employs the DROP heuristic, has been introduced for single-file databases and compared with the ADD heuristic. In an extensive test performed for its validation, the DROP heuristic found optimal solutions in all cases. In comparison, the ADD heuristic found nonoptimal solutions in several occasions.

The index selection algorithm using the DROP heuristic has been extended to incorporate the clustering property (Algorithms 2 and 3) and also has been extended for application to multiple-file databases (Algorithm 4).

Although index selection has long been a subject of intensive research, no successfully validated algorithm with good efficiency has been reported. We believe that our approach provides a useful and reliable algorithm for practical applications.

Appendix G. Relationships between Relations

In this section, we demonstrate that the assumption that we made in Appendix A excluding M-to-N relationships from consideration for optimization is reasonable.

Relations can have various relationships (not necessarily semantically meaningful ones) depending on the characteristics of the domains of the attributes that are related. For example, if we relate a key attribute (or set of attributes) in relation R_1 and a nonkey attribute (or set of attributes) in relation R_2, then R_1 and R_2 have a 1-to-N relationship with respect to these attributes considered. Relations R_1 and R_2 will have a 1-to-1 relationship if attributes considered in both relations are key attributes, and an M-to-N relationship if both are nonkey attributes.

In this section, we shall show that a relation scheme any of whose relation instance is a join of two relations which has an M-to-N relationship with respect to a set of attributes A has a multivalued dependency (MVD) [ULL 82]—assuming that the only predicate that relates these two relations is the one that represents the join on A.

Intuitively, if a relation scheme R has an MVD $A \rightarrow\rightarrow B$ (and accordingly $A \rightarrow\rightarrow R-B$), where A and B are sets of attributes in R, then in a specific relation instance r of R, given a specific value of A, the values of $R-B$ are completely replicated for every distinct value of B. Because of this replication, sets of attributes B and $R-B$ tend not to have a meaningful relationship, and thus it does not make much sense to have both sets of attributes together in a single relation.

We believe, in accordance with the above argument, that joining two relations that have M-to-N relationships with respect to the set of attributes on which the join is performed is relatively infrequent. In Appendix A, on the basis of this argument, we excluded from consideration as prospects for optimization join operations on relations that bear an M-to-N relationship.

We have the following theorems:

Theorem G.1: If a relation scheme R has an MVD $A \rightarrow\rightarrow B$, where A and B are sets of attributes of R, then every relation r for R is a natural join of projections of r on the relation schemes $R_1 = A$, $R_2 = A \cup B$, $R_3 = A \cup (R - B)$, respectively, where R_1, R_2, and R_3 possess the relationships shown in Figure G-1.

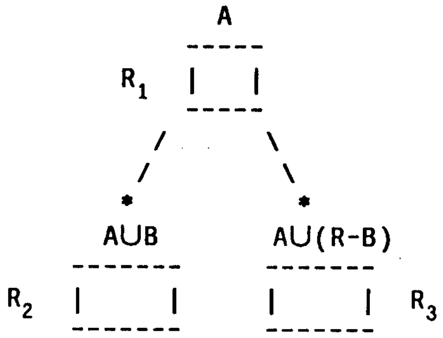


Figure G-1: Relation Schemes and Their Relationships.

In this figure – – * represents a 1-to-N relationship with respect to A.

Proof: R_1, R_2, and R_3 can be obtained by two consecutive lossless join decompositions, i.e., decomposition of R into $A \cup B$ and $A \cup (R-B)$ and decomposition of $A \cup B$ into A and $A \cup B$. These two decompositions are lossless, since we have an MVD $A \rightarrow\rightarrow B$ [ULL 82]. Thus, the overall join decomposition of R into R_1, R_2, and R_3 is also lossless. Therefore, for any relation r for R, $r = \mathrm{JOIN}_{i=1}^{3}\Pi_{R_i}(r)$.

To prove that R_1 and R_2 has a 1-to-N relationship, we note that A in R_1 is a key, since it is the only attribute (or set of attributes) in R_1. However, A in R_2 is generally not a key. So we have a 1-to-N relationship from R_1 to R_2.

When A in R_2 is a key, we have a 1-to-1 relationship between R_1 and R_2, which can be considered as a special case of a 1-to-N relationship. Similarly, R_1 and R_3 have a 1-to-N relationship. Q.E.D.

Theorem G.2: A relation scheme R has MVDs $A \rightarrow\rightarrow B$ and $C \rightarrow\rightarrow D$ if any relation r for R is a natural join of some relations r_1, r_2, and r_3 for relation schemes R_1, R_2, and R_3, respectively, where R_1, R_2, and R_3 have the relationships shown in Figure G-2.

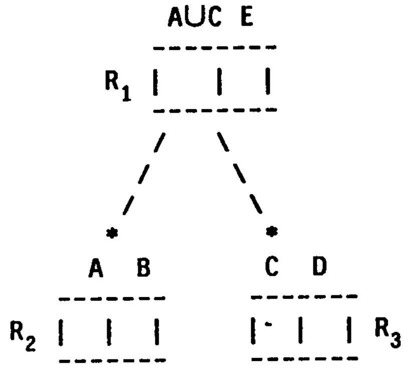


Figure G-2: Relation Schemes and Their Relationships.

In that figure – –* represents a 1-to-N relationship with respect to A on the left side and one with respect to C on the right side.

Proof: Consider tuples t and s with $t[A] = s[A]$ in a relation r for R. Since r is a natural join of some relations r_1, r_2, and r_3, respectively, there must exist tuples u_1, u_2 in r_1; v_1, v_2 in r_2; and w_1, w_2 in r_3 such that

$$t[A] = u_1[A] = v_1[A] \text{ and } t[C] = u_1[C] = w_1[C]$$
$$s[A] = u_2[A] = v_2[A] \text{ and } s[C] = u_2[C] = w_2[C].$$

Since $t[A] = s[A]$, we have $u_1[A] = u_2[A]$. But since R_1 and R_2 have a 1-to-N relationship from R_1 to R_2, and they are connected through A, A must have unique values in r_1. Hence $u_1 = u_2$ and accordingly $u_1[C] = u_2[C] = w_2[C]$.

Therefore r will contain a tuple z where

$$z[A] = v_1[A] = t[A] = s[A]$$
$$z[B] = v_1[B] = t[B]$$
$$z[R-A\cup B] = w_2[R\text{-}A\cup B] = s[R-A\cup B].$$

Thus R has an MVD A→→B. By a similar argument, R has C→→D. Q.E.D.

Corollary: Let relation schemes R_1 and R_2 have an M-to-N rclationship with respect to a set of attributes A. The relation scheme R whose relation instances are natural joins on A of two relations r_1 for R_1 and r_2 for R_2 has MVDs $A \rightarrow\rightarrow (R_1 - A)$ and $A \rightarrow\rightarrow (R_2 - A)$.

Proof: We can consider a two-relation join of r_1 and r_2 as a three-relation join of r_1, r_2, and an imaginary relation $\Pi_A r_1 \cup \Pi_A r_2$. Then the relation scheme R_3 corresponding to this imaginary relation has 1-to-N relationships with R_1 and R_2, with respect to A, as shown in Figure G-3.

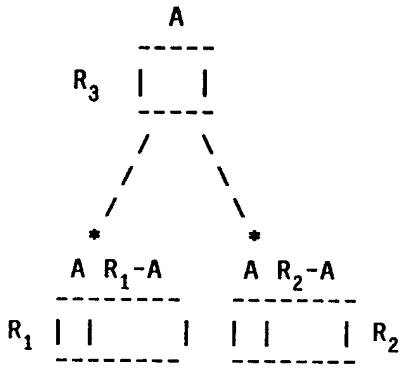


Figure G-3: Relation R_3 has 1-to-N Relationships with R_1 and R_2.

Thus relation scheme R has MVDs $A \rightarrow\rightarrow (R_1 - A)$ and $A \rightarrow\rightarrow (R_2 - B)$ from Theorem G.2. Q.E.D.

Appendix H. Equivalent Restriction Frequency of a Partial-Join

In Appendix A, the *equivalent restriction frequency* of a partial-join using the join index method was defined as the ratio of the gain in access cost by having the restriction indexes in a partial-join to the gain in access cost that the same restriction indexes would yield in the joint restriction with the join index. We shall show in this section that this equivalent restriction frequency of a partial join using the join index method performed on relation R_2 can be calculated, with one exceptional case, as Cf_{12}/F_a, where Cf_{12} is the coupling factor from relation R_1 to relation R_2 and F_a is the selectivity of the join columns of relation R_2.

By formulating the partial-join cost and the cost of the joint restriction in both cases in which the restriction index is used and in which the restriction index is not used (or does not exist), we shall show that the number of block accesses saved in a partial-join is the same as the number of block accesses saved in the joint restriction of the join index and the restriction index used in the partial-join multiplied by Cf_{12}/F_a.

We have three general cases: in Case 1 both the join index and the restriction index are nonclustering; in Case 2 the join index is nonclustering, while the restriction index is clustering; in case 3 the join index is clustering, while the restriction index is nonclustering.

Case 1: both the join index and the restriction index are nonclustering

a. When the restriction index is used
Joint restriction cost = $b(m,p,F_a \times F_i \times n)$
Partial-join cost = $(Cf_{12}/F_a)\, b(m,p,F_a \times F_i \times n)$

In a joint restriction, the number of records selected is $F_a \times F_i \times n$. We assume that these records are evenly spread and are accessed in TID order. Thus we get $b(m,p,F_a \times F_i \times n)$ block accesses. In a partial-join, we are following the join index in the order of join column value, and $F_a \times F_i \times n$ records are accessed for a distinct join column value. Since these records are spread over the entire file and are accessed in TID order, we get $b(m,p,F_a \times F_i \times n)$ block accesses. This procedure is repeated for every distinct join column value selected by the coupling effect and the join selectivity (i.e., according to the coupling factor). The total number of distinct join column values are $1/F_a$. Therefore, as the partial-join cost, we have $(Cf_{12}/F_a)\ b(m,p,F_a \times F_i \times n)$.

b. When the restriction index is not used (or does not exist)

Joint restriction cost = $b(m,p,F_a \times n)$

Partial-join cost = $(Cf_{12}/F_a)\ b(m,p,F_a \times n)$

An analysis applies that is the same as above except that the restriction index is not used. Thus, we have $F_a \times n$ selected records instead of $F_a \times F_i \times n$.

Case 2: the join index is nonclustering while the restriction index is clustering

There are two cases to be considered separately: when $F_i \times m \geq 1$ and when $F_i \times < 1$.

1. When $F_i \times m \geq 1$

a. When the restriction index is used

Joint restriction cost = $b(F_i \times m,\ p,\ F_{\times a} F_i \times n)$

Partial-join cost = $(Cf_{12}/F_a)\ b(F_i \times m,\ p,\ F_a \times F_i \times n)$.

This case is almost identical to Case 1, except that the restriction index is clustering and the range within which the selected records can be found is limited to $F_i \times m$ blocks instead of m (the number of blocks of the entire file). To use b function it is required that $F_i \times m \geq 1$.

b. When the restriction index is not used (or does not exist)

Joint restriction cost = $b(m,p,F_a \times n)$

Partial-join cost = $(Cf_{12}/F_a)\ b(m,p,F_a \times n)$

This case is exactly the same as Case 1-b.

2. When $F_i \times m < 1$

a. When the restriction index is used

Joint restriction cost = $F_i \times b(1/F_i, F_i \times n, F_a \times n)$

Partial-join cost = $(Cf_{12}/F_a) \times F_i \times b(1/F_i, F_i \times n, F_a \times n)$.

Since $F_i \times m < 1$ and the restriction index is clustering, all records selected according to the restriction index will be confined in an area smaller than 1 block (let us call this a *selected area*). Let us assume that this selected area resides within a physical block (i.e., we ignore the case in which this selected area resides on the border of two blocks). If we assume that the file is divided into logical blocks of the same size as this selected area, the probability that this selected area will be hit by a joint restriction is

$(1/(1/F_i))\ b(1/F_i, F_i \times n, F_a \times n)$.

This is also the probability that the physical block containing the selected area will be hit (note that there are $1/F_i$ logical blocks in the file). This is also the number of physical blocks to be hit by the joint restriction, since the physical block containing the selected area is the only one that can possibly be accessed.

In a partial-join, the same analysis is valid for each distinct join column value, assuming that the same block must be fetched again if a repeated forward scan inside this block is to be performed. Thus the partial-join cost is the product of (Cf_{12}/F_a) and the joint restriction cost.

b. When the restriction index is not used

Joint restriction cost = $b(m,p,F_a \times n)$

Partial-join cost = $(Cf_{12}/F_a)\ b(m,p,F_a \times n)$

This case is exactly the same as Case 1-b.

Case 3: the join index is clustering, while the restriction index is nonclustering

1. When $F_a \times m \geq 1$

a. When the restriction index is used

Joint restriction cost = $b(F_a \times m, p, F_a \times F_i \times n)$

Partial-join cost = $(Cf_{12}/F_a)\, b(F_a \times m, p, F_a \times F_i \times n)$.

An analysis similar to Case 2-1-a applies, except that the range of the selected records is limited to $F_a \times m$ blocks instead of $F_i \times m$.

b. When the restriction index is not used

Joint restriction cost = $F_a \times m$

Partial-join cost = $(Cf_{12}/F_a) \times F_a \times m = Cf_{12} \times m$.

Since the join index is clustering, the number of blocks accessed is proportional to the number of records selected.

2. When $F_a \times m < 1$

a. When the restriction index is used

Joint restriction cost = $(1/(1/F_a))\, b(1/F_a, F_a \times n, F_i \times n)$

Partial-join cost = $b(m, 1/(F_a m), Cf_{12} \times b(1/F_a, F_a \times n, F_i \times n))$.

The joint restriction cost can be obtained by a similar analysis used in Case 2-2-a, except that the roles of F_a and F_i are interchanged.

In the partial-join, the entire file is divided into $1/F_a$ logical blocks, each of which contains $F_a \times n$ records. According to the restriction index, $F_i \times n$ records are selected; the number of logical blocks selected by this restriction is $b(1/F_a, F_a \times n, F_i \times n)$.

The coupling factor Cf_{12} determines how many distinct join column values are actually selected.

Since one logical block corresponds to one distinct join column value, the number of logical blocks selected according to the coupling factor and the selectivity of the restriction index is $Cf_{12} \times b(1/F_a, F_a \times n, F_i \times n)$.

To calculate the number of physical blocks hit, let us assume that the entire file consists of m blocks, each of which contains $1/(F_a m)$ logical blocks. Since $Cf_{12} \times b(1/F_a, F_a \times n, F_i \times n)$ logical blocks are selected, the number of physical blocks that will be hit is $b(m, 1/(F_a m), Cf_{12} \times b(1/F_a, F_a \times n, F_i \times n))$.

b. When restriction index is not used (or does not exist)
Joint restriction cost = 1
Partial-join cost = $b(m,1/(F_a m),Cf_{12}/F_a)$

This can be easily derived from Case 3-2-b by setting F_i to 1.

We have seen, in all situations except Case 3-2, that the partial-join cost is equivalent to Cf_{12}/F_a times the joint restriction cost. Accordingly, the cost saved by having the restriction index in a partial-join is Cf_{12}/F_a times the cost saved by having the restriction index in the joint restriction.

Case 3-2 is the only case in which the equivalent restriction frequency of a partial-join using the join index method cannot be represented as Cf_{12}/F_a. The reason is that, in a partial-join, the logical blocks are accessed in a serial order, and thus several logical blocks may cause only one block access. In the case of joint restriction, we need one block access in any case if at least one record is selected.

The derivations of the formulas were introduced to show how we can formulate cost formulas with the b function, as well as to show that, in most cases, equivalent restriction frequency has a simple form, Cf_{12}/F_a.

While the detailed form of cost formulas depend on the specific cost models, we believe that the same principle we used in the derivation can be easily applied to any given model.

Appendix I. Computational Errors

I.1 Comparison of Computational Errors

In this appendix we develop the prediction of the computational errors which occur in the estimation of block accesses discussed in Appendix C. These computational errors occur due to the limited precision of the computing system evaluating the formula.

For convenience, we reintroduce two equations from Appendix C. Equation (I.1) is the approximation formula developed, and Equation (I.3) is the representation of Yao's exact formula using the Gamma function.

$$\begin{aligned} b_{w1}(m,p,k)/m = {} & [1 - (1-1/m)^{k}] && \text{(I.1)} \\ & + [1/m^{2}p \times k(k-1)/2 \times (1-1/m)^{k-1}] && \\ & + [1.5/m^{3}p^{4} \times k(k-1)(2k-1)/6 \times (1-1/m)^{k-1}] && \\ & \qquad\qquad \text{when } k \leq n-p, \text{ and} && \\ b_{w1}(m,p,k)/m = {} & 1 \qquad\qquad \text{when } k > n-p && \text{(I.2)} \\ b(m,p,k) = {} & m[1 - \exp(\mathrm{LGAM}(n\text{-}p+1) + \mathrm{LGAM}(n\text{-}k+1) && \text{(I.3)} \\ & \quad - \mathrm{LGAM}(n\text{-}p\text{-}k+1) - \mathrm{LGAM}(n+1))]. && \end{aligned}$$

Theorem I.1: Calculation of Eq. (I.3) to d digits of precision with a possible error of ±1 in the least significant digit (LSD) requires at least $\log_{10}(mn \log(n)) + d$ valid digits in the computing system with a possible error of ±1 in the LSD.

Proof: We shall use a pseudo equality symbol $\doteq$ throughout this proof and the proof of Theorem I.2, ignoring the deviation from equality whenever it neither affects the logical flow of the proof nor changes the numerical result significantly.

By Stirling's approximation [KNU-a 73],

$$\Gamma(n+1) = \sqrt{2\pi n}(n/e)^{n}, \text{ and}$$

$$\ln(\Gamma(n+1)) = \ln(\sqrt{2\pi}) + 0.5 \ln(n) + n(\ln(n) - 1)$$

$$\doteq n \ln(n),$$

since we are considering relatively large n's.

From Eq. (I.3),

$$b(m,p,k)=1\text{-}\exp[LGAM(n\text{-}p+1)+LGAM(n\text{-}k+1)\text{-}LGAM(n\text{-}p\text{-}k+1)\text{-}LGAM(n+1)] \tag{I.4}$$
$$\doteq \text{-}LGAM(n\text{-}p+1)\text{-}LGAM(n\text{-}k+1)+LGAM(n\text{-}p\text{-}k+1)+LGAM(n+1).$$

Let us consider the case in which $k=1$. At this k value, all four terms in Eq. (I.4) are close to n ln(n), the result is the smallest possible, and we shall get the maximum error. If we assume that evaluation of Eq. (I.4) causes the error of ±1 in the LSD, then the error of the result will be

$$10^{-x} \times n \ln(n),$$

where x is the number of significant digits.

The exact value of the result of Eq. (I.4) must be 1/m, since only one block will be hit. Therefore, the relative error caused by the computation with x significant digits will be

$$(10^{-x} \times n \ln(n))/(1/m)=(mn \ln(n)) \times 10^{-x}. \tag{I.5}$$

If we require this to have an error of less than 10^{-d}, so that we have d digits of precision in the result with a possible error of ±1 in the LSD, Eq. (I.5) must be less than 10^{-d}. Therefore,

$$x \geq \log_{10}(mn \ln(n)) + d. \qquad \text{Q.E.D.}$$

Theorem I.2: $x \geq (\log_{10} m)+d+\log_{10}(d)+1$ valid digits with a possible error of ±1 in the LSD are sufficient in the calculation of Eq. (I.1) to d digits of precision.

Proof: The major cause of the error is in the calculation of $1-1/m$ as m gets larger, since it requires as many digits as $\log_{10}$ m. We shall use the equality $(1-1/m)^m=e^{-1}$ throughout, assuming that m is sufficiently large. For convenience let us consider only the first term of Eq. (I.1), since the other terms behave similarly and their absolute values are always less than $(1-1/m)^k$.

Let us divide the values of k into 3 ranges: $k < 0.1m$, $k > \ln(10)\times d\times m$, and $0.1m \leq k \leq \ln(10)\times d\times m$.

(1) $k < 0.1m$

From a Taylor expansion we have

$$(1-1/m)^k=1-k/m+k(k-1)/2 \times (1/m)^2 \ldots \doteq 1-k/m, \text{ and thus}$$
$$1-(1-1/m)^k \doteq k/m.$$

In the calculation of $(1-1/m)$ we have an error of 10^{-x}, so that, as a result of computation, we get

$$(1-1/m+10^{-x})^k \doteq 1-k(1/m- 10^{-x}).$$

(For convenience let us consider only a positive error. Negative errors can be treated similarly.)

Accordingly, the error of the overall calculation will be

$$(k(1/m- 10^{-x})-k/m)/(k/m)=-10^{-x} \times m.$$

Thus, we get a precision of d digits in the result if and only if

$$10^{-x} \times m < 10^{-d}, \text{ or}$$
$$x \geq (\log_{10} m)+d.$$

(2) $k > \ln(10) \times d \times m$

In this case $0 < (1-1/m)^k < 10^{-d}$. Hence,

$$1 > 1-(1-1/m)^k > 1-10^{-d} \geq 0.9,$$

assuming $d \geq 1$. However, actual computation may yield

$$1-(1-1/m+10^{-x})^k.$$

Since

$$x \geq (\log_{10} m)+d+1,$$

we have

$$10^{-x} \leq (1/m)10^{-(d+1)}.$$

Since

$$\begin{aligned}&(1-1/m+10^{-(d+1)}/m)^k\\ &= (1-(1-10^{-(d+1)})/m)^k\\ &\leq (1-(1-10^{-(d+1)})/m)^{\ln(10)\times d\times m}\\ &= 10^{-(1-10^{-(d+1)})\times d} \doteq 10^{-d}\end{aligned}$$

assuming $d \geq 1$, the relative error, $((1-1/m)^k - (1-1/m+10^{-x})^k)/0.9$, cannot be greater than $(1/0.9)(10^{-d}) \doteq 10^{-d}$. Thus we have a precision of d digits in the result.

(3) $0.1m \leq k \leq \ln(10) \times d \times m$

We have

$$\begin{aligned}
&\ln[((1-1/m)+10^{-x})/(1-1/m)^{k}]\\
&= k(\ln(1-1/m+10^{-x})-\ln(1-1/m))\\
&\doteq k((1-1/m+10^{-x})-(1-1/m))\\
&= k \times 10^{-x}.
\end{aligned}$$

Accordingly,

$$\begin{aligned}
&((1-1/m+10^{-x})^{k}-(1-1/m)^{k})/(1-1/m)^{k}\\
&\doteq \exp(k\times 10^{-x}) - 1.
\end{aligned}$$

a) $m \leq k \leq (\ln 10)\times d \times m$

The relative error will be

$$\begin{aligned}
&((1-1/m+10^{-x})^{k}-(1-1/m)^{k})/(1-(1-1/m)^{k})\\
&< ((1-1/m+10^{-x})^{k}-(1-1/m)^{k}))/(1-1/m)^{k}\\
&\doteq \exp(k\times 10^{-x}) - 1\\
&\leq \exp((k/md)\times 10^{-(d+1)}) - 1\\
&\leq \exp(\ln(10)\times 10^{-(d+1)}) - 1\\
&< \ln(10)\times 10^{-(d+1)}\\
&= 0.23\times 10^{-d}.
\end{aligned}$$

Thus, we have a precision of d digits.

b) $0.1\ m \leq k \leq m$

We have

$$(1-1/m)^{k} \doteq 1-k/m \leq 0.9.$$

Hence, the relative error will be

$$\begin{aligned}
&((1-1/m+10^{-x})^{k}-(1-1/m)^{k})/(1-(1-1/m)^{k})\\
&\leq (1/0.1)((1-1/m+10^{-x})^{k}-(1-1/m)^{k})\\
&\leq 10((1-1/m+10^{-x})^{k}-(1-1/m)^{k})/(1-1/m)^{k}\\
&\doteq 10(\exp(k\times 10^{-x}) - 1)\\
&\leq 10(\exp((k/m)\times 10^{-(d+1)}) - 1)\\
&\leq 10((k/m)\times 10^{-(d+1)})
\end{aligned}$$

$$\leq 10 \times 10^{-(d+1)}$$
$$= 10^{-d}.$$

This shows that we have a precision of d digits. Q.E.D.

Corollary: Eq. (I.1) requires at least $x \geq (\log_{10} m)+d$ valid digits to get d digits of precision in the result.

Proof: This follows from the case (1) of Theorem I.2. Q.E.D.

Applying Theorem I.2 and its corollary, the actual requirement will be

$$(\log_{10} m)+d \leq x \leq (\log_{10} m)+d+\log_{10}(d)+1.$$

Example 1: Let us calculate the number of valid digits required by the evaluation of Eq. (I.1) and Eq. (I.3), respectively, when $m=10^6$, $p=10$, $n=10^7$, and we need a precision of 2 digits in the result.

(a) For Eq. (I.1),

$\log_{10}(10^6)+2+\log_{10}(2)+1=9.3$,
$\log_{10}(10^6)+2=8$, and
$8 \leq x \leq 9.3$.

(b) For Eq. (I.3),

$$x=\log_{10}(10^6 \times 10^7 \times \ln(10^7)) + 2$$
$$= 16.3.$$

We note that Eq. (I.3) requires roughly twice as many valid digits as does Eq. (I.1). □

In the exhaustive calculation we made over the range specified in Appendix C, the maximum error (0.2%) occurred at $m=10^6$, $p=1$, and $k \ll m$ (i.e. $k \simeq 1$), which actually corresponds to the lower bound given in the corollary.

Example 2: The error of 0.2% is equivalent to a precision of 2 digits according to our definition, since 0.998 compared with 1.0 clearly has an error exceeding 1 in the third digit, and the first and second digits are the only valid digits with possible error of ±1 in the LSD. Thus, the number of valid digits x of the computer required by Eq. (I.1)) when $m = 10^6$ will be

$$8 \leq x \leq 9.3$$

The DECSYSTEM-20 has 2^{-27} of resolution, approximately corresponding to 8 valid digits, which confirms our result. □

I.2 Computational Error in an Extended Range

The maximum computational error when the number of blocks m is extended to 10^7 is 4.3%; it occurs at $k = 1$ for all values of p.

We assumed throughout that m has only integer values. However, computer calculation performed over all combinations of the following range shows that the maximum deviation of Eq. (I.1) from the exact formula is 3.7%, even for the real values of m.

- $1.1 \leq p \leq 3.9$ with increments of 0.1,
- $1 \leq p \leq 10$ where p is an integer,
- $1.1 \leq m \leq 3.9$ with increments of 0.1.

The general shape of the deviation can be found in Appendix C.

Appendix J. Supplementary Discussions on Design Algorithms

This appendix consists of six sections. In Section J.1 more details are presented on the development of physical design algorithms that have not been fully discussed in Chapter 4. In Section J.2 are discussed more details on the strategy of handling virtual columns (multiattribute columns). In Section J.3 more complete formulas for the time complexities of the Index Selection Step and the Exhaustive-Search Method are derived. The two situations that produced deviations in the tests are analyzed in Section J.4.

J.1 More Details on Design Algorithms

In this section we discuss some details on the development of the design algorithms that have not fully explained in Chapter 4. Specifically, we have the following four fine details to discuss:

1. An index together with the clustering property

In the Clustering Design Step (or NS Clustering Design Step), an index is assigned together with the clustering property if the column has not been assigned one in the Index Selection Step (or NS Index Selection Step). If the column has an index already, only the clustering property is assigned. This strategy has been used on the basis of the observation that in almost none of optimal solutions a column possesses the clustering property without an index (except for degenerate cases in which multiple optimal solutions exist). This observation confirms the belief that the clustering property is best utilized when it is coupled with an index. Furthermore, although there is nothing wrong in having a clustering column without an index in the access configuration as far as it is one of the optimal solutions, having such a column during the design process could hinder smooth transitions of access configurations resulting in a nonoptimal solution. These considerations support the strategy of assigning an index together any time the clustering property is assigned to a column.

2. Index selection before clustering design

The Index Selection Step must precede the Clustering Design Step in the iteration loop. In the preliminary algorithm introduced in [WHA-a 81], clustering design is performed before index selection. However, doing so posed the following problem. If the Index Selection Step precedes the Clustering Design Step, clustering design is performed with a full index set in the first iteration. As a result the total index update cost, which constitutes a major portion of the total update cost, stays the same whichever column acquires the clustering property. Thus, in determining the optimal clustering column, there is a possibility that the clustering property is assigned to a column that is heavily updated. The problem is that, in the Index Selection Step performed next, the column endowed with the clustering property (which has a heavy update cost) has a tendency to release neither the clustering property nor the index even though an index is not worth its update cost because an index coupled with the clustering property yields much more benefit than the index alone so that the index may look like worth of its own update cost. The result would be a wrong index and a wrong clustering column. Furthermore, this mistake won't be corrected in future iterations.

We can avoid this anomaly by swapping the order of the two design steps. We start with index selection assuming no clustering column initially. Since no indexes are coupled with the clustering property, all the indexes can be compared on a fair basis. The indexes that do not compensate for their own update cost will subsequently be dropped. The clustering property is assigned in the next step when all insignificant indexes have been dropped. Note that although we swapped the two steps, the same problem can arise since the Index Selection Step follows the Clustering Design Step of the previous iteration; but it would be much less hazardous than in the first iteration.

3. All join methods allowed in the first iteration

In Phase 1 of Algorithm 1 and 2, during the first iteration, we allow all join methods for update transactions. In Section D.3.2 it was shown that, in Phase 1, only the join index method can be used for update transactions having join predicates. This restriction led to an anomaly that the join indexes used by update transactions must not be removed in Phase 1. We greatly alleviated this anomaly by introducing the Perturbation Step. However, further improvement can be achieved by releasing the constraint during the first iteration so that other join methods may be used as well, and the join indexes for update transactions can be dropped. Note that this strategy is not logically correct but is only a temporary measure to make a smooth flow of design process. The constraint is imposed again from the second iteration.

4. Calculation of the selectivity

The selectivity of a range predicate (that has an operator such as <, <=, >, >=) is arbitrarily set to 1/4. A more elaborate method for estimating the selectivity of a range predicate would be to interpolate based on the highest and the lowest values in the column and the value specified in the predicate. However, specific methods of estimating the selectivity of a range predicate does not affect the general validity of the design algorithms. Therefore, in this dissertation, a most simplistic method is employed.

J.2 Virtual Columns

In this section we discuss the strategy of handling virtual columns. Virtual columns are necessary to support indexes defined on two or more attributes (multiattribute indexes). The general treatment of multiattribute indexes adds another level of complexity to the already complex index selection problem. (A very simplified version of the index selection problem has been proved to be NP-complete by Comer [COM 78].) Moreover, in the context of our model, multiattribute indexes are not necessary for resolving restriction predicates. Restriction predicates referring to more than one attribute can always be resolved by forming the intersection of TID sets from the indexes involved to the same effect of a multiattribute index. (Let us remember that we have assumed that TID manipulation causes negligible I/O accesses.)

When a join predicate that refers to more than one attribute is resolved, however, individual single-attribute indexes cannot be used as a substitute for a multiattribute index: join indexes must be scanned according to the order of join column values, but this order cannot be achieved by simply intersecting the single-attribute indexes that are defined for the join attributes. Thus, in principle, we need to consider a multiattribute index for every set of join attributes that appear together in a join predicate. Each set of join attributes are subsequently mapped into a virtual column.

Sets of attributes constituting virtual columns are specified in the schema information. Virtual columns are defined only for the sets of attributes that are semantically relevant as join attributes. The concept of *connections* and *connecting attributes* is borrowed from the *Structural Model*[WIE 79] for this purpose. The structural model defines the *connection* as the representation of a semantically meaningful relationship between two relations, and *connecting attributes* as the attributes establishing the relationship that corresponds to a connection on the basis of equality of their values. We define *semantically relevant joins* as those associated with connections. Accordingly, the connecting attributes of a connection are mapped into a virtual column. Let us note that, in evaluating the joins that are not semantically relevant but have more than one join attributes, the join index method cannot be used since virtual columns and accordingly multiattribute indexes are not provided for their join attributes.

In Figure J-1 below, is illustrated a simple database schema with connections and virtual columns as well as ordinary single-attribute columns. The symbol "*" in a connection indicates N-side relation in the 1-to-N relationship that the connection represents.

Another purpose of defining a virtual column is to provide a correct selectivity of an equality

```
 ---------------------
 | SHIPID | SHIPNAME |                    SHIPS
 ---------------------
    |
    |
    *   VOYAGES
 ------------------------------------
 | SHIPID | VOYAGENUMBER | CHARTERER |    VOYAGES
 ------------------------------------
 \_________   __________/
          \/
          |
          |
          *
  _________/\___________
 /                      \
 -------------------------------------
 | SHIPID | VOYAGENUMBER | STOPNUMBER |   STOPS
 -------------------------------------
```

Columns:

In Relation SHIPS : SHIPID, SHIPNAME
In Relation VOYAGES : SHIPID, VOYAGENUMBER, CHARTERER
In Relation STOPS : SHIPID, VOYAGENUMBER, STOPNUMBER

Virtual Columns:

In Relation VOYAGES : SHIPID-VOYAGENUMBER
In Relation STOPS : SHIPID-VOYAGENUMBER

Figure J-1: Relations, Connections, Columns, and Virtual Columns.

predicate referring to more than one column. If the predicate refers to only one column, the selectivity is estimated as the inverse of the column cardinality (the number of distinct values existing in a column). If the predicate refers to more than one column, i.e., if the predicate is a conjunction of more than one simple equality predicate that refers to a single column, the selectivity is often estimated as the product of inverses of column cardinalities of the columns referred in the predicate (let us call the set of these columns the *column set*). Such an estimation is valid, however, only under the assumption that there is no correlation among the columns [SCH 75]. This assumption implies that every possible combination of distinct values from individual columns in the column set must exist in the database. This assumption is obviously impractical in most cases.

When a restriction predicate is considered, however, we extend the assumption as follows: For each nonexistent value combination, a hypothetical tuple is created, and it is assumed that the predicates applied to the column set select each tuple of distinct value (including hypothetical tuples) with equal probability. If the predicate selects hypothetical tuples, the response will be null. This assumption is further elaborated in Example A.1.

Example A.1: Let us assume that we have the following data for a column set (A, B, C). The data represent the projection of a relation on the column set. Duplicates are removed so that unique value combinations are represented by one tuple in the projection.

Column Set	A	B	C
	a_1	b_1	c_1
Data	a_1	b_2	c_1
	a_2	b_1	c_2
	a_2	b_2	c_2

If hypothetical tuples are included, the data for the column set become

Column Set	A	B	C	
	a_1	b_1	c_1	
	a_1	b_1	c_2	hypothetical
Data	a_1	b_2	c_1	
	a_1	b_2	c_2	hypothetical
	a_2	b_1	c_1	hypothetical
	a_2	b_1	c_2	
	a_2	b_2	c_1	hypothetical
	a_2	b_2	c_2	

Then, the assumption states that the predicate of the form (A = 'a') AND (B = 'b') AND (C = 'c') refers to each (distinct) tuple in the data including hypothetical tuples with equal probability. Thus, the probability that the value combination $\langle a_2, b_2, c_1 \rangle$ is to be accessed is 1/8. □

The above assumption is an extension of the uniformity assumption applied to individual columns; the uniformity assumption asserts that the equality predicate referring to a column selects each distinct value in that column with equal probability and that there exist an equal number of tuples for each distinct value. Under this extended uniformity assumption the joint selectivity of a column set becomes the product of the selectivities of the component columns. (For simplicity, we define the selectivity of the column as the selectivity of an equality predicate for a column.)

Although the extended uniformity assumption is useful for estimating the joint selectivity of a restriction predicate, it cannot easily be applied when a join predicate is concerned. When a join operation is performed, values of join attributes from each relation are compared for a possible match. Hence, the join predicate is only tested with the join attribute values that actually exist in the database; that is, hypothetical tuples are never selected. For this reason, the probability of an existing tuple to be selected is far greater than what would result from the extended uniformity assumption. This phenomenon is further illustrated in Example A.2.

Example A.2: Consider the following relation:

Attributes:	EMP-NAME	CHILD-NAME	AGE
	John Meadows	Jack	3
	John Meadows	Alby	5
Data	John Meadows	Sara	7
	Charlie Fu	Randy	5
	Charlie Fu	David	10

The column cardinality for EMP-NAME is 2; that for CHILD-NAME is 5. Thus, the selectivity of the column EMP-NAME is 1/2; that for CHILD-NAME is 1/5. If EMP-NAME and CHILD-NAME are referred together in a restriction predicate, the joint selectivity of the two columns is 1/10. This is so because it is conceivable that a user specifies a predicate such as (EMP-NAME = 'John Meadows') AND (CHILD-NAME = 'Randy'). If the two columns are specified together in a join operation (i.e., EMP-NAME and CHILD-NAME are the join attributes), however, a predicate such as (EMP-NAME = 'John Meadows') AND (CHILD-NAME = 'Randy') are never tested since all the values of the columns are supplied from within the database. Thus, the joint selectivity of the two columns is 1/5, the inverse of the cardinality of the virtual column (EMP-NAME, CHILD-NAME). □

So far, it has been shown that joint selectivities are different according to the type of predicates we consider. The difference can be reflected in the design process by specifying a selectivity for each virtual column. Thus, the virtual column and its selectivity will be used when a multiattribute join is performed, whereas selectivities of individual columns will be used when a multiattribute restriction is resolved.

During the design process a virtual column is considered as yet another independent column without any difference from an ordinary column. There is one exception, however, when the clustering property is assigned to a virtual column. When a relation is sorted according to the order of a multiattribute column, it is also sorted according to the order of its first component column. Thus, if a virtual column is assigned the clustering property, so should its first component column be, but not vice versa.

J.3 More Details on Time Complexities

In this section we provide a more detailed derivation of the time complexity of the Index Selection Step in Algorithm 1 and Algorithm 2. The time complexity of NS Index Selection Step in Algorithm 3 can be derived similarly. The time complexity of the Exhaustive-Search Algorithm is also presented.

1. Index Selection Step

The following notation will be used throughout this section:

v_0: Number of columns in a relation (number of indexes in a full index set)

v_1: Number of indexes remaining when the index selection substep with $k=1$ (corresponding to substeps 2, 3, and 4 of the Index Selection Step in Section D.5.1) has been completed. k is the maximum number of indexes that have been considered together at a time.

v_i: Number of indexes remaining when the index selection substep with $k=i$ has been completed.

s: Average number of relations referred to in a transaction.

t: Number of transactions in the input usage information.

c: Total number of columns in the database

c_i: Number of columns in relation i.

Let us first consider the time complexity of the index selection substep with $k=1$. During the first iteration in the substep, the algorithm tries to drop one index at a time, actually dropping the one that yields the maximum benefit. Thus $(s/r) \times t \times v_0$ calls to the cost evaluator (EVALCOST-1) are necessary. The factor $(s/r) \times t$ takes into account that, on the average, $(s/r) \times t$ transactions refers the relation being considered. During the second iteration only $(s/r) \times t \times (v_0 - 1)$ calls are necessary since an index has been already dropped in the first iteration. As a result, for the entire substep with $k=1$, the number of calls to the cost estimator will be

$$(s/r) \times t \times (v_0 + (v_0 - 1) + (v_0 - 2) + \ldots + v_1). \tag{J.1}$$

Now let us consider the substep with $k=2$. This substep starts with v_1 indexes that survived the substep with $k=1$. During the first iteration the algorithm tries to drop every possible pair among v_1 indexes; hence, the cost evaluator will be called $(s/r)t(v_1(v_1-1)/2)$ times. For the second iteration, it will be called $(s/r)t(v_1-2)(v_1-3)/2$ times. Thus, for the entire substep with $k=2$, the number of calls to the cost evaluator will be

$$0.5\,(s/r)t(v_1(v_1-1) + (v_1-2)(v_1-3) + (v_1-4)(v_1-5) + \ldots + v_2(v_2-1)). \tag{J.2}$$

The complexities for higher values of k can be obtained analogously.

As we see in Equations ((J.1)) and ((J.2)), the time complexities have a dynamic nature in that they depend on the number of indexes remaining after each substep. In general, however, the complexity of the first substep is $O((t/r)v_0^2)$ and that of the second substep is $O((t/r)v_0^3)$ since v_1 will be roughly proportional to v_0. Analogously, complexities for higher values of k, in general, would be $O((t/r)v_0^{k+1})$. Since a higher order substep has a higher order complexity, the complexities of lower order steps become negligible as v_0 gets larger. Thus, the overall complexity of the Index Selection Step is $O((t/r)v_0^{k+1})$.

2. Exhaustive-Search Algorithm -

The time complexity of searching all the possible alternative access configuration for the entire database is obtained as

$$t\times(c_1+1)(c_2+1)...(c_r+1)\times 2^c. \tag{J.3}$$

The factor 2^c accounts for the number of possible index configuration since every column in the database can either have an index or not. The factor (c_i+1) represents the number of possible clustering positions in relation i (including the case with no clustering column). The number of possible index configuration multiplied by the number of clustering positions in every relation will constitute the total number of access configurations for the entire database. For each access configuration, the cost evaluator (EVALCOST-2) will be called by the number of transactions, t, in the usage information. Thus, Equation (J.3) gives the total number of calls to the cost evaluator for searching through all the alternative access configurations.

J.4 Analysis of Deviations

In most situations that are tested in Section K, all three algorithms produced optimal solutions. In some cases, however, some deviations occurred from the optimal solutions: Algorithm 1 produced a deviation of 3.1% in Situation 50; Algorithm 1 and Algorithm 3 produced 6.6% in Situation 42. In this section these situations are investigated and the deviation analyzed.

The following notation will be used throughout this section:

1: A clustering column with an index
0: A column with an index only
X: A column with neither an index nor the clustering property
$\overset{x}{1}$: A column with the clustering property but no index

1. Algorithm 1 in Situation 50

Figure J-2 shows access configurations for relations R_2 and R_5 at each design step of Algorithms

1, 2, and 3. Access configurations of the other relations are not shown since they are identical to the optimal solution. Only first two iterations are shown since there are no more improvements from the third.

Relation	R2												R5					
Column	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C1	C2	C1	C2
Design Step	Alg.1				Alg.2				Alg.3				Alg.1		Alg.2		Alg.3	
Index Sel.	0	0	0	0	N/A				0	0	0	0	0	X	N/A		X	X
Cluster Des.	0	0	0	1	0	0	0	1	0	0	0	1	1	X	1	X	X	1
Perturbation	X	0	0	1	X	0	0	1	N/A				1	X	1	X	N/A	
Index Sel.	X	0	0	1	N/A				0	0	0	1	1	X	N/A		X	X
Cluster Des.	X	0	0	1	X	0	0	1	0	0	0	1	1	X	X	1	X	1
Perturbation	X	0	0	1	0	0	0	1	N/A				1	X	X	1	N/A	

Figure J-2: Access Configurations for R_2 and R_5 at Each Design Step.

In this situation Algorithm 2 and Algorithm 3 both found the optimal solution. Algorithm 1, however, resulted in a slight deviation from the optimal solution. Compared with the optimal solution, the access configuration that Algorithm 1 produced has the clustering property on $R_5.C_1$ instead of $R_5.C_2$ and lacks an index on $R_2.C_1$.

R_5 has the clustering property on C_1 because, in the Index Selection Step during the first iteration, an index has been assigned to C_1. The column C_1 subsequently acquired the clustering property since the configuration (1 X) is less costly than (0 1), and the same configuration stayed until the algorithm terminated. Since the clustering design is performed in a separate step in Algorithm 1, the configuration (X 1), which is less costly than (1 X) cannot be reached without passing through (0 1). Thus, the deviation in this situation is partially due to the separation of the Index Selection Step and the Clustering Design Step, i.e., vertical partitioning.

Let us note, however, that the error situation occurs because R_5 obtains an index on C_1 in the first Index Selection Step. (In comparison, Algorithm 3 does not assign an index to $R_5.C_1$, even though it also has separate Index Selection and Clustering Design Steps.) The index is assigned on C_1 because it is more beneficial to use the join index method for Transaction 5 than to use the sort-merge method. The inner/outer-loop join method – the one used in the optimal solution – cannot be used in the index selection step because of the conditions for separability. Thus, horizontal partitioning also partially caused the deviation. We note that Algorithms 2 and 3, which utilize only one type of partitioning, do not produce any deviation.

The index on $R_2.C_1$ is dropped by Algorithm 1 since, in processing Transaction 5, it is more beneficial to use the inner/outer-loop join method with the join direction from R_2 to R_5 and to drop the index on $R_2.C_1$ than to use the join index method while retaining that index. The inner/outer-loop join for R_2 to R_5 has an advantage especially because $R_5.C_1$ has the clustering property.

The access configuration produced by Algorithm 1 is only slightly different from the optimal solution. Accordingly, when the frequencies of the transactions are changed in Situations 51 and 52, this deviation disappears and all three algorithms find the optimal solution.

2. Algorithm 1 and Algorithm 3 in Situation 42

The deviation in this situation occurred due to a very peculiar reason that the access configurations (0 0 X) for relation DOCKS yields the exactly same cost as those of (1 0 X) and (0 1 X). (Optimal solutions are (1 X X) and (X 1 X).) The access configuration (0 0 X) is obtained from the Index Selection Step of the first iteration. Since, in the next step (Clustering Design Step) the clustering property is assigned only if there is nonzero improvement in the cost, the clustering property cannot be assigned. (That is, neither (1 0 X) nor (0 1 X) does not yield positive improvement in the cost compared with (0 0 X).)

If the clustering property were assigned to any one of the first two columns, the other of the two would be dropped in the Index Selection Step of the next iteration yielding an optimal solution.

This deviation could be made somewhat significant: by deliberately adjusting the frequency of Transaction 12 up to 175, a deviation of 27.2% has been observed. However, since the mechanism that caused this error is very peculiar, it is believed that the chance of the mechanism being invoked is negligible when more transactions acting upon relation DOCKS are added in the usage. (The chance that two different access configurations have the exactly same cost is very slim.) Also, when a large database is considered, the local deviation caused by this mechanism might well be just a small portion of the entire cost, so that the relative deviation of the entire design may be negligible.

Appendix K. The Physical Database Design Optimizer – An Implementation

In this section we introduce the Physical Database Design Optimizer(PhyDDO) which implements the three design algorithms. Also, a set of input situations that have been tested to validate the design algorithms and their results are presented.

As in Appendix J.4, the following notation will be used throughout this appendix:

1: A clustering column with an index
0: A column with an index only
X: A column with neither an index nor the clustering property
$\overset{x}{1}$: A column with the clustering property but no index

The PhyDDO is an experimental system to develop various heuristics for the physical database design. Besides the three design algorithms, implemented in the PhyDDO are the Exhaustive-Search Algorithm and the one-shot evaluator that simply evaluates the cost of the access configuration initially given by the user. The latter has been proved to be an effective tool for debugging the system. The system accepts eight types of transactions:

SQ: Single-relation (one-variable) queries

JQ: Two-relation (two-variable) queries having join predicates (i.e., two-relation joins)

AQ: Single-relation queries having aggregate operators in their SELECT clauses, or GROUP BY constructs [CHA 76] or both. (A type AQ transaction is essentially a partial-join between the GROUP BY column and the relation itself as far as the I/O access cost is concerned.)

SU: Single-relation update transactions.

JU: Update transactions having join predicates.

SD: Single-relation deletion transactions.

JD: Deletion transactions having join predicates.

INS: Insertion transactions (single-relation transactions only).

Transactions are specified together with their types and frequencies as the input usage information. On the other hand, the schema and the data characteristics for the database are specified as the input schema information. An example of complete input information for a database consisting of two relations is presented in Figure K-1.

```
SCHEMA
    RELATIONS
        RELATION R1
            RELCARD 50
            NBLOCKS 10
            BLKFAC  5
            COLUMN C1
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 1
                INDEX     1

            COLUMN C2
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0

            COLUMN C3
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


        RELATION R2
            RELCARD       1000
            NBLOCKS       100
            BLKFAC        10
            COLUMN        C1
                COLCARD   1000
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


            COLUMN        C2
                COLCARD   7
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


            COLUMN        C3
                COLCARD   50
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 1
                INDEX     1


    CONNECTIONS

        CONNECTION
            REL1        R1
            COL1        C1
            JSEL1       1.0
            RELN        R2
            COLN        C3
            JSELN       1.0


USAGE
    TRANSACTION 1
        TYPE SQ         FREQ 100
        SELECT R1.C1, R1.C3
        FROM   R1(0.5)
        WHERE  R1.C2 = "ANY" AND
               R1.C1 = 3

    TRANSACTION 2
        TYPE JQ         FREQ 50
        SELECT R1.C1, R2.C1
        FROM   R1(0.3), R2(0.3)
        WHERE  R1.C1 = R2.C3 AND
               R2.C1 > 500

    TRANSACTION  3
        TYPE SU         FREQ 10
        UPDATE R1
        SET    R1.C3 = "ANY"
        WHERE  R1.C2 = "ANY"

    TRANSACTION  4
        TYPE JU         FREQ 10
        UPDATE R2
        SET     R2.C2 = R2.C2 + 1
        FROM    R2(1), R1(0.3)
        WHERE   R2.C3 = R1.C1 AND
                R1.C3 = "ANY" AND
                R2.C2 > 5

    TRANSACTION  5
        TYPE SD          FREQ 10
        DELETE R2
        WHERE   R2.C2 >= 7

    TRANSACTION  6
        TYPE JD          FREQ 10
        DELETE R1
        FROM    R1, R2(0.2)
        WHERE   R1.C1 = R2.C3 AND
                R2.C2 > 7

    TRANSACTION  7
        TYPE INS         FREQ 10
        INSERT INTO R2:
                <1001, "ANY", 20000, 3, 1>
@
```

Figure K-1: An input specification for PhyDDO.

The keywords used in the schema and usage specification in Figure K-1 are explained below:

Relcard: Number of tuples in a relation (relation cardinality).

Nblocks: Number of disk blocks a relation occupies.

Blkfac: Number of tuples in one disk block (blocking factor).

Colcard: Number of distinct values in a column (column cardinality).

Niblk: Number of disk blocks that an index would occupy if it existed.

Iblkfac: Number of index entries in one disk block (index blocking factor).

Clustered: 1 if the column is clustered in the initial access configuration given by the user; 0 otherwise. If not explicitly specified, the default is 0.

Index: 1 if the index exists in the initial access configuration given by the user; 0 otherwise. If not explicitly specified, the default is 0.

Mcolumn: A multiattribute column in a relation (virtual column).

Components: Component columns of a multiattribute column.

Rell: The relation on the 1-side of 1-to-N relationship represented by a connection (Relation 1).

Coll: Connecting attribute of Relation 1. A virtual column if there are more than one connecting attribute.

Jsell: Ratio of the number of distinct join column values participating in the unconditional join (a join without restriction predicate) to the total number of distinct join column values or the ratio of the number of nondangling tuples to the total number of tuples (join selectivity).

RelN: The relation on the N-side of 1-to-N relationship represented by a connection (Relation N).

ColN: Connecting attributes of Relation N.

JselN: Join Selectivity for Relation N.

Type: Transaction type.

Freq: Relative frequency of a transaction.

As outputs, the system produces the optimal access configuration of the database and the total processing cost. It also produces optimal join methods and their costs for two-variable transactions.

Twenty one different input situations have been tested to validate the heuristics used in the design algorithms. The input situations tested consist of seven schemas, each schema being accompanied by three variations of usage specification. First, the transactions and their frequencies are defined so that by intuition they look most natural. Second, according to the test result with the first usage specification, the frequencies are modified so that the costs of transactions are roughly of the same order. This modification prevents a few most costly transactions from dominating the results of the design. Third, all the queries are eliminated from the usage specification leaving only update transactions. This modification simulates a situation where there are heavy updates.

Described in Figures K-2 to K-8 are all the tested input situations as they are submitted to the Physical Database Design Optimizer together with their optimal solutions. Each input situation is named as Situation ij where $i \in (1,2,3,4,5,6,7)$ shows which schema is used, and $j \in (0,1,2)$ which variation of the usage information is used. To simply the illustrations, for each schema, three situations with different usage specification have been merged into one figure: in the usage specification, relative frequencies from three situations are specified in the same row in the order of j; in the illustration of the optimal solutions (except for Situations 70, 71, and 72) three solutions are presented from the top in the order of j, using the notation introduced in Appendix J. For Situations 70, 71, and 72, optimal solutions are presented in text form since they are too big to be drawn in a figure.

A copy of the source code for PhyDDO and an executable file are stored in <kbms> PhyDDO.pas and PhyDDO.exe at SRI-AI. The LALR syntax description of the usage and schema information (including the syntax of the transactions supported) can be found in <kbms> PhyDDO.grammar.

```
SCHEMA
    RELATIONS
        RELATION R1
            RELCARD 50
            NBLOCKS 10
            BLKFAC  5
            COLUMN C1
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 1
                INDEX     1

            COLUMN C2
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0

            COLUMN C3
                COLCARD   50
                NIBLK     1
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


        RELATION R2
            RELCARD       1000
            NBLOCKS       100
            BLKFAC        10
            COLUMN        C1
                COLCARD   1000
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


            COLUMN        C2
                COLCARD   7
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 0
                INDEX     0


            COLUMN        C3
                COLCARD   50
                NIBLK     5
                IBLKFAC   200
                CLUSTERED 1
                INDEX     1


    CONNECTIONS

        CONNECTION
            REL1        R1
            COL1        C1
            JSEL1       1.0
            RELN        R2
            COLN        C3
            JSELN       1.0


USAGE
    TRANSACTION 1
        TYPE SQ         FREQ 100        1000         Deleted
        SELECT R1.C1, R1.C3
        FROM   R1(0.5)
        WHERE  R1.C2 = "ANY" AND
               R1.C1 = 3

    TRANSACTION 2
        TYPE JQ         FREQ 50         50           Deleted
        SELECT R1.C1, R2.C1
        FROM   R1(0.3), R2(0.3)
        WHERE  R1.C1 = R2.C3 AND
               R2.C1 > 500

    TRANSACTION  3
        TYPE SU         FREQ 10         100          100
        UPDATE R1
        SET    R1.C3 = "ANY"
        WHERE  R1.C2 = "ANY"

    TRANSACTION  4
        TYPE JU         FREQ 10         100          100
    UPDATE R2
    SET    R2.C2 = R2.C2 + 1
    FROM   R2(1), R1(0.3)
    WHERE  R2.C3 = R1.C1 AND
           R1.C3 = "ANY" AND
           R2.C2 > 5

TRANSACTION  5
    TYPE SD         FREQ 10         100          100
    DELETE R2
    WHERE  R2.C2 >= 7

TRANSACTION  6
    TYPE JD         FREQ 10         100          100
    DELETE R1
    FROM   R1, R2(0.2)
    WHERE  R1.C1 = R2.C3 AND
           R2.C2 > 7

TRANSACTION  7
    TYPE INS        FREQ 10         1000         1000
    INSERT INTO R2:
           <1001, "ANY", 20000, 3, 1>
```

SCHEMA1X

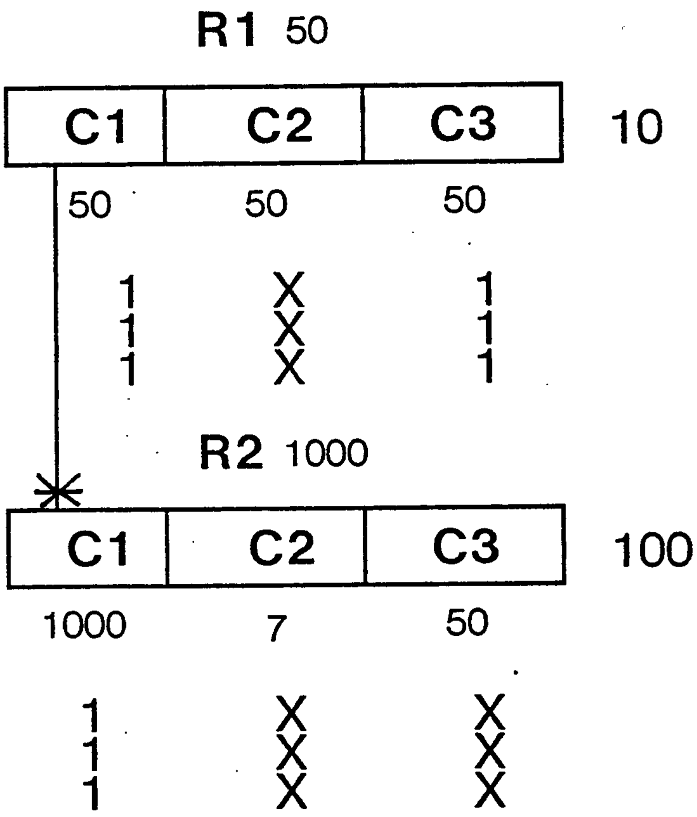


Figure K-2: Situations 10, 11, 12, and their optimal solutions.

```
SCHEMA
    RELATIONS
        RELATION COUNTRIES
            RELCARD 100
            NBLOCKS 20
            BLKFAC 5

            COLUMN COUNTRYNAME
                COLCARD         100
                NIBLK           1
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN POPULATION
                COLCARD         100
                NIBLK           1
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

        RELATION SHIPS
            RELCARD     1000
            NBLOCKS     100
            BLKFAC      10

            COLUMN ID
                COLCARD         1000
                NIBLK           5
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN S←COUNTRY
                COLCARD         30
                NIBLK           5
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

        RELATION VOYAGES
            RELCARD             100000
            NBLOCKS             5000
            BLKFAC              20

            COLUMN SHIP←ID
                COLCARD         1000
                NIBLK           500
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN VOYAGENO
                COLCARD         200
                NIBLK           500
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN CHARTERER
                COLCARD         10000
                NIBLK           500
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

        RELATION SHIP←CHARTERER
            RELCARD             10000
            NBLOCKS             500
            BLKFAC              20

            COLUMN      C←NAME
                COLCARD         10000
                NIBLK           50
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN      C←COUNTRY
                COLCARD         100
                NIBLK           50
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

    CONNECTIONS
        CONNECTION
            REL1        COUNTRIES
            COL1        COUNTRYNAME
            JSEL1       0.3
            RELN        SHIPS
            COLN        S←COUNTRY
            JSELN       1.0
        CONNECTION
            REL1        SHIPS
            COL1        ID
            JSEL1       1.0
            RELN        VOYAGES
            COLN        SID
            JSELN       1.0

        CONNECTION
            REL1        SHIP←CHARTERER
            COL1        C←NAME
            JSEL1       1.0
            RELN        VOYAGES
            COLN        CHARTERER
            JSELN       1.0

        CONNECTION
            REL1        COUNTRIES
            COL1        COUNTRYNAME
            JSEL1       1.0
            RELN        SHIP←CHARTERER
            COLN        C←COUNTRY
            JSELN       1.0
USAGE
    TRANSACTION 1
        TYPE    SQ      FREQ    100     100     Deleted
        SELECT  COUNTRIES.POPULATION
        FROM    COUNTRIES(0.3)
        WHERE   COUNTRIES.COUNTRYNAME = "USA"

    TRANSACTION 2
        TYPE    SQ      FREQ    100     100     Deleted
        SELECT  SHIPS.S←COUNTRY
        FROM    SHIPS(0.3)
        WHERE   SHIPS.ID = 101

    TRANSACTION 3
        TYPE    JQ      FREQ    20      20      Deleted
        SELECT  SHIPS.ID, COUNTRIES.POPULATION
        FROM    SHIPS(0.3), COUNTRIES(0.3)
        WHERE   SHIPS.S←COUNTRY = COUNTRIES.COUNTRYNAME AND
                SHIPS.ID = 101

    TRANSACTION 4
        TYPE    JQ      FREQ    20      20      Deleted
        SELECT  SHIP←CHARTERER.C←NAME, COUNTRIES.POPULATION
        FROM    SHIP←CHARTERER(0.3), COUNTRIES(0.5)
        WHERE   SHIP←CHARTERER.C←COUNTRY = COUNTRIES.COUNTRYNAME  AND
                SHIP←CHARTERER.C←NAME = "SMITH←TRADING←CO"

    TRANSACTION 5
        TYPE    JQ      FREQ    50      10      Deleted
        SELECT  VOYAGES.CHARTERER, VOYAGES.SID, SHIPS.S←COUNTRY
        FROM    SHIPS(0.5), VOYAGES(0.5)
        WHERE   SHIPS.ID = VOYAGES.SID AND
                VOYAGES.CHARTERER = "SMITH←TRADING←CO"

    TRANSACTION 6
        TYPE    JQ      FREQ    50      2       Deleted
        SELECT  VOYAGES.SID, VOYAGES.VNUMBER, VOYAGES.CHARTERER,
                SHIP←CHARTERER.C←COUNTRY
        FROM    VOYAGES(0.5), SHIP←CHARTERER(0.5)
        WHERE   VOYAGES.CHARTERER = SHIP←CHARTERER.C←NAME  AND
                VOYAGES.SID = 17

    TRANSACTION 7
        TYPE    SU      FREQ    5       5       5
        UPDATE  COUNTRIES
        SET     COUNTRIES.POPULATION = 35000000
        WHERE   COUNTRIES.COUNTRYNAME = "KOREA"

    TRANSACTION 8
        TYPE    SD      FREQ    100     100     100
        DELETE  VOYAGES
        WHERE   VOYAGES.SID = 51 AND
                VOYAGES.CHARTERER = "SMITH←TRADING←CO"

    TRANSACTION 9
        TYPE    INS     FREQ    10      1000    1000
        INSERT INTO SHIPS:
                <1051, "ANY←COUNTRY">

    TRANSACTION 10
        TYPE    SD      FREQ    50      1       1
        DELETE  SHIP←CHARTERER
        WHERE   SHIP←CHARTERER.C←COUNTRY = "USSR"

    TRANSACTION 11
        TYPE    JU      FREQ    0       1       1
        UPDATE  SHIPS
        SET     SHIPS.S←COUNTRY = "BIG←COUNTRY"
```

```
FROM    SHIPS, COUNTRIES
WHERE   SHIPS.S←COUNTRY = COUNTRIES.COUNTRY←NAME  AND
        COUNTRIES.POPULATION > 100000000  AND
        SHIPS.ID = 100
```

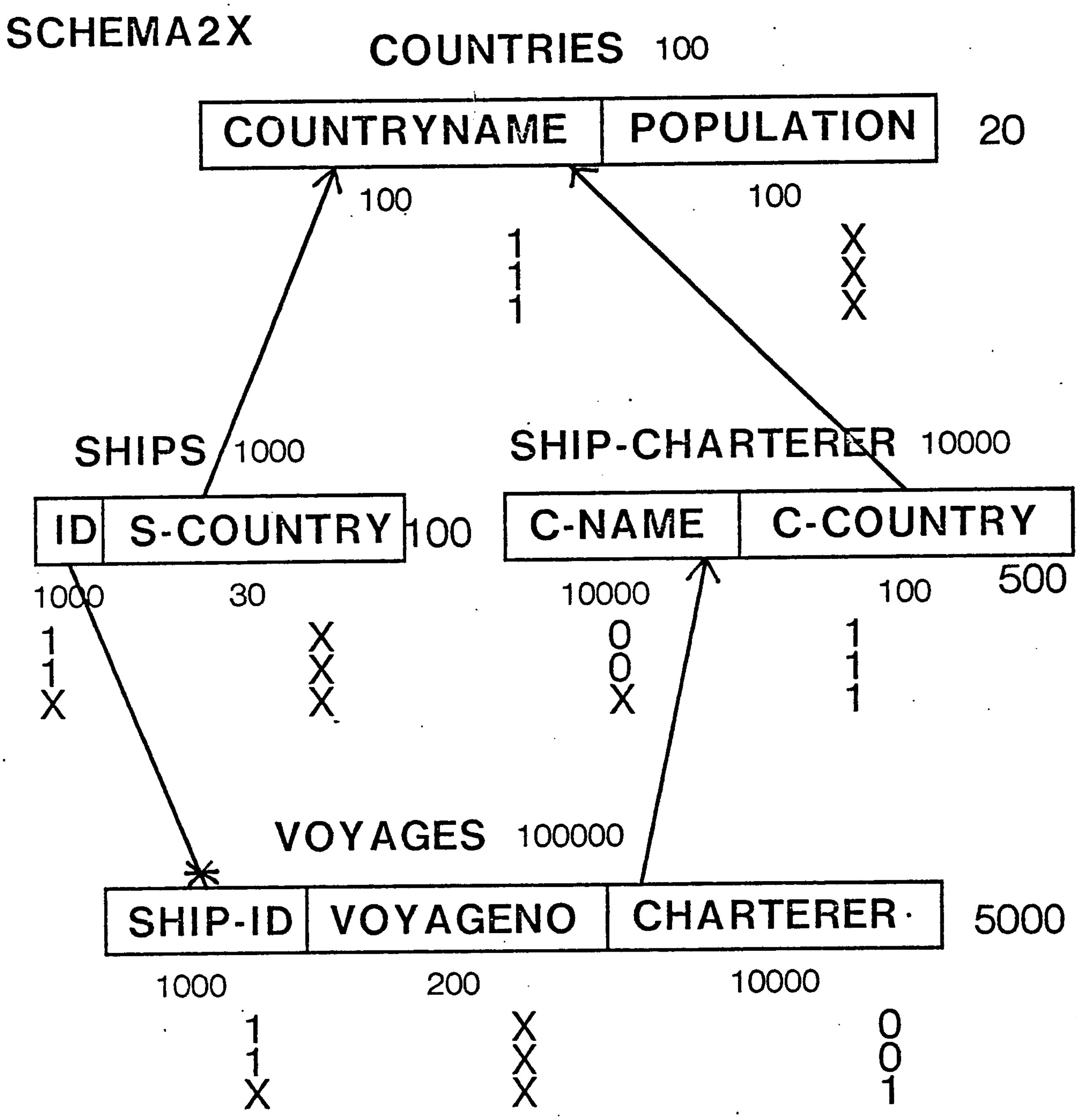


Figure K-3: Situations 20, 21, 22, and their optimal solutions.

```
SCHEMA
    RELATIONS
        RELATION            DEPTS
            RELCARD         100
            NBLOCKS         20
            BLKFAC          5

            COLUMN          DEPTNO
                COLCARD             100
                NIBLK               1
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          LOCATION
                COLCARD             20
                NIBLK               1
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

        RELATION            EMPS
            RELCARD                 10000
            NBLOCKS                 1000
            BLKFAC                  10

            COLUMN          EMPNO
                COLCARD             10000
                NIBLK               50
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          DEPTNO
                COLCARD             100
                NIBLK               50
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          JOB
                COLCARD             100
                NIBLK               50
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          SALARY
                COLCARD             10000
                NIBLK               50
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

        RELATION            CHILDREN
            RELCARD                 20000
            NBLOCKS                 1000
            BLKFAC                  20

            COLUMN          EMPNO
                COLCARD             10000
                NIBLK               100
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          NAME
                COLCARD             20000
                NIBLK               100
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          AGE
                COLCARD             20
                NIBLK               100
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

        RELATION            EMP←PROJ
            RELCARD                 20000
            NBLOCKS                 500
            BLKFAC                  40

            COLUMN          EMPNO
                COLCARD             10000
                NIBLK               100
                IBLKFAC             200
                CLUSTERED           0
                INDEX               1

            COLUMN          PROJNO
```

```
                COLCARD         100
                NIBLK           100
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

            COLUMN      PERCENT←TIME
                COLCARD         100
                NIBLK           100
                IBLKFAC         200
                CLUSTERED       0
                INDEX           1

    CONNECTIONS
        CONNECTION
            REL1        DEPTS
            COL1        DEPTNO
            JSEL1       1.0
            RELN        EMPS
            COLN        DEPTNO
            JSELN       1.0

        CONNECTION
            REL1        EMPS
            COL1        EMPNO
            JSEL1       1.0
            RELN        CHILDREN
            COLN        EMPNO
            JSELN       1.0

        CONNECTION
            REL1        EMPS
            COL1        EMPNO
            JSEL1       1.0
            RELN        EMP←PROJ
            COLN        EMPNO
            JSELN       1.0

USAGE
    TRANSACTION 1
        TYPE    SQ      FREQ    100     100     Deleted
        SELECT  DEPTS.LOCATION
        FROM    DEPTS
        WHERE   DEPTS.DEPTNO = 15

    TRANSACTION 2
        TYPE    AQ      FREQ    20      20      Deleted
        SELECT  EMPS.DEPTNO, EMPS.JOB, AVG(EMPS.SALARY)
        FROM    EMPS(0.5)
        WHERE   EMPS.JOB = "WELDER"
        GROUP BY EMPS.DEPTNO

    TRANSACTION 3
        TYPE    SQ      FREQ    100     100     Deleted
        SELECT  EMP←PROJ.EMPNO, EMP←PROJ.PROJNO, EMP←PROJ.PERCENT←TIME
        FROM    EMP←PROJ
        WHERE   EMP←PROJ.EMPNO = 293

    TRANSACTION 4
        TYPE    JQ      FREQ    20      5       Deleted
        SELECT  EMPS.DEPTNO, EMPS.EMPNO, EMP←PROJ.PROJNO
        FROM    EMPS(0.3), EMP←PROJ(0.3)
        WHERE   EMPS.EMPNO = EMP←PROJ.EMPNO   AND
                EMP←PROJ.PROJNO = 11

    TRANSACTION 5
        TYPE    JQ      FREQ    20      200     Deleted
        SELECT  DEPTS.LOCATION
        FROM    DEPTS(0.5), EMPS(0.3)
        WHERE   DEPTS.DEPTNO = EMPS.DEPTNO  AND
                EMPS.EMPNO = 55

    TRANSACTION 6
        TYPE    JQ      FREQ    10      10      Deleted
        SELECT  EMPS.DEPTNO, EMPS.EMPNO
        FROM    EMPS(0.3), CHILDREN(0.2)
        WHERE   EMPS.EMPNO = CHILDREN.EMPNO AND
                EMPS.JOB = "WELDER"  AND
                CHILDREN.AGE < 10

    TRANSACTION 7
        TYPE    SU      FREQ    50      50      50
        UPDATE  EMPS
        SET     EMPS.SALARY = EMPS.SALARY + 1000
        WHERE   EMPS.JOB = "WELDER"

    TRANSACTION 8
        TYPE    JU      FREQ    10      10      10
        UPDATE  EMP←PROJ
        SET     EMP←PROJ.PERCENT←TIME = 30
        FROM    EMP←PROJ, EMPS(0.3)
        WHERE   EMPS.EMPNO = EMP←PROJ.EMPNO  AND
                EMPS.DEPTNO = 5  AND
```

```
              EMP←PROJ.PROJNO = 17

TRANSACTION 9
    TYPE     SD       FREQ    10       2        2
    DELETE   EMPS
    WHERE    EMPS.DEPTNO = 3

TRANSACTION 10
    TYPE     SD       FREQ    10       10       10
    DELETE   DEPTS
    WHERE    DEPTS.DEPTNO = 3

TRANSACTION 11
    TYPE     JD       FREQ    5        2        2
    DELETE   EMPS
    FROM     EMPS, EMP←PROJ(0.3)
    WHERE    EMPS.EMPNO = EMP←PROJ.EMPNO   AND
             EMP←PROJ.PROJNO = 5

TRANSACTION 12
    TYPE     SD       FREQ    10       100      100
    DELETE   CHILDREN
    WHERE    CHILDREN.EMPNO = 175
```

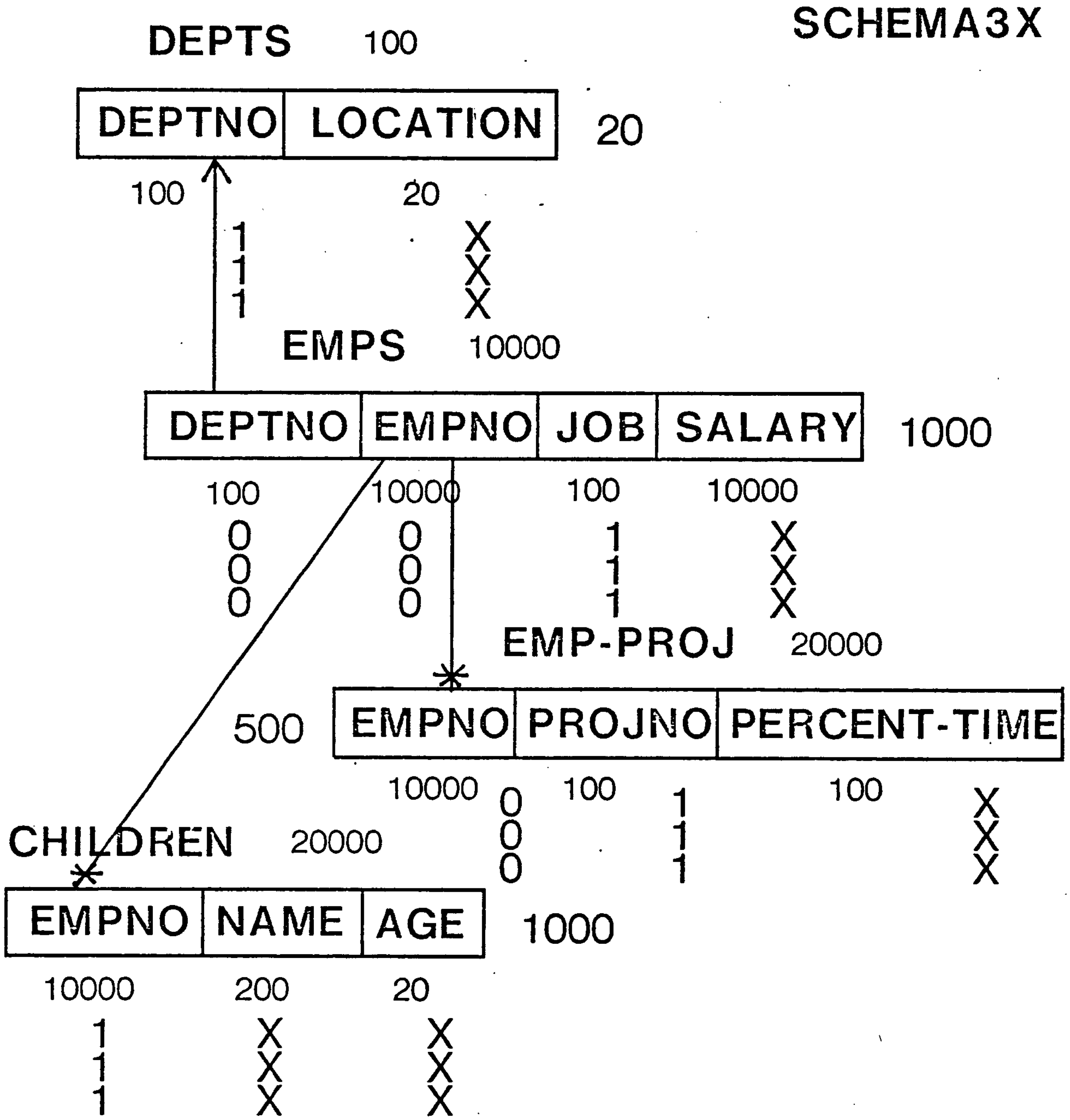


Figure K-4: Situations 30, 31, 32, and their optimal solutions.

```
SCHEMA
    RELATIONS
        RELATION        PORTS
            RELCARD     1000
            NBLOCKS     200
            BLKFAC      5

            COLUMN      P←NAME
                COLCARD         1000
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      NUM←SHIPS
                COLCARD         20
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      NUM←WHOUSE
                COLCARD         50 !MAX 50 WAREHOUSES/PORT!
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        DOCKS
            RELCARD     5000
            NBLOCKS     1000
            BLKFAC      5

            COLUMN      P←NAME
                COLCARD         1000
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      DOCKNO
                COLCARD         100 !DOCK←NUMBER DOES NOT HAVE TO BE NUMBERED
                                     CONTIGUOUSLY!
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN   SHIP←ID
                COLCARD         3000 !NOT EVERY DOCK HAS A SHIP ANCHORED!
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        WAREHOUSES
            RELCARD     10000 !10 WAREHOUSED/PORT ON THE AVERAGE!
            NBLOCKS     2000
            BLKFAC      5

            COLUMN      P←NAME
                COLCARD         1000
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      WHOUSENO
                COLCARD         200
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      CARGOCLASS
                COLCARD         100
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        CARGOCLASSES
            RELCARD     100
            NBLOCKS     50
            BLKFAC      2

            COLUMN      CARGOCLASS
                COLCARD         100
                NIBLK           1
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1
            COLUMN      W←UNIT
                COLCARD         50
                NIBLK           1
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1


    CONNECTIONS
        CONNECTION
            REL1        PORTS
            COL1        P←NAME
            JSEL1       1.0
            RELN        DOCKS
            COLN        P←NAME
            JSELN       1.0

        CONNECTION
            REL1        PORTS
            COL1        P←NAME
            JSEL1       1.0
            RELN        WAREHOUSES
            COLN        P←NAME
            JSELN       1.0

        CONNECTION
            REL1        CARGOCLASSES
            COL1        CARGOCLASS
            JSEL1       1.0
            RELN        WAREHOUSES
            COLN        CARGOCLASS
            JSELN       1.0


USAGE
    TRANSACTION 1
        TYPE    SQ      FREQ    100     50      Deleted
        SELECT  DOCKS.SHIP←ID
        FROM    DOCKS
        WHERE   DOCKS.P←NAME = "PORT←A"  AND
                DOCKS.DOCKNO = 3

    TRANSACTION 2
        TYPE    SQ      FREQ    100     50      Deleted
        SELECT  DOCKS.P←NAME, DOCKS.DOCKNO
        FROM    DOCKS
        WHERE   DOCKS.SHIP←ID = 101

    TRANSACTION 3
        TYPE    SQ      FREQ    100     50      Deleted
        SELECT  PORTS.NUM←SHIPS
        FROM    PORTS
        WHERE   PORTS.P←NAME = "PORT←A"

    TRANSACTION 4
        TYPE    JQ      FREQ    20      10      Deleted
        SELECT  PORTS.P←NAME, PORTS.NUM←WHOUSE, WAREHOUSE.WHOUSENO,
                WAREHOUSES.CARGOCLASS
        FROM    PORTS(0.5), WAREHOUSES(1)
        WHERE   PORTS.P←NAME = WAREHOUSES.P←NAME  AND
                PORTS.P←NAME = "PORT←A"

    TRANSACTION 5
        TYPE    JQ      FREQ    20      2       Deleted
        SELECT  WAREHOUSES.P←NAME, WAREHOUSES.WHOUSENO
        FROM    WAREHOUSES(0.5), CARGOCLASSES(0.3)
        WHERE   WAREHOUSES.CARGOCLASS = CARGOCLASSES.CARGOCLASS  AND
                CARGOCLASSES.W←UNIT = "GALLON"

    TRANSACTION 6
        TYPE    SU      FREQ    100     50      50
        UPDATE  PORTS
        SET     PORTS.NUM←SHIPS = PORTS.NUM←SHIPS + 1
        WHERE   PORTS.P←NAME = "PORT←A"

    TRANSACTION 7
        TYPE    SU      FREQ    1       50      50
        UPDATE  PORTS
        SET     PORTS.NUM←WHOUSE = PORTS.NUM←WHOUSE + 1
        WHERE   PORTS.P←NAME = "PORT←A"

    TRANSACTION 8
        TYPE    SQ      FREQ    50      5       Deleted
        SELECT  WAREHOUSES.P←NAME, WAREHOUSES.WHOUSENO
        FROM    WAREHOUSES
        WHERE   WAREHOUSES.CARGOCLASS = "EXPLOSIVES"

    TRANSACTION 9
        TYPE    JQ      FREQ    20      20      Deleted
        SELECT  PORTS.P←NAME, PORTS.NUM←SHIPS, DOCKS.DOCKNO, DOCKS.SHIP←ID
        FROM    PORTS(0.5), DOCKS(1)
        WHERE   PORTS.P←NAME = DOCKS.P←NAME  AND
                PORTS.P←NAME = "PORT←A"
```

```
TRANSACTION 10
    TYPE    SU      FREQ    100     100     100
    UPDATE  DOCKS
    SET     DOCKS.SHIP←ID = 101
    WHERE   DOCKS.P←NAME = "PORT←A"  AND
            DOCKS.DOCKNO = 3

TRANSACTION 11
    TYPE    INS     FREQ    1       30      30
    INSERT  INTO    CARGOCLASSES:
            <"FROZEN←FISH", "TON">

TRANSACTION 12
    TYPE    INS     FREQ    1       20      20
    INSERT  INTO    DOCKS:
            <"PORT←A", 7, 0>

TRANSACTION 13
    TYPE    INS     FREQ    1       20      20
    INSERT  INTO    WAREHOUSES:
            <"PORT←A", 15, "FROZEN←FISH">
```

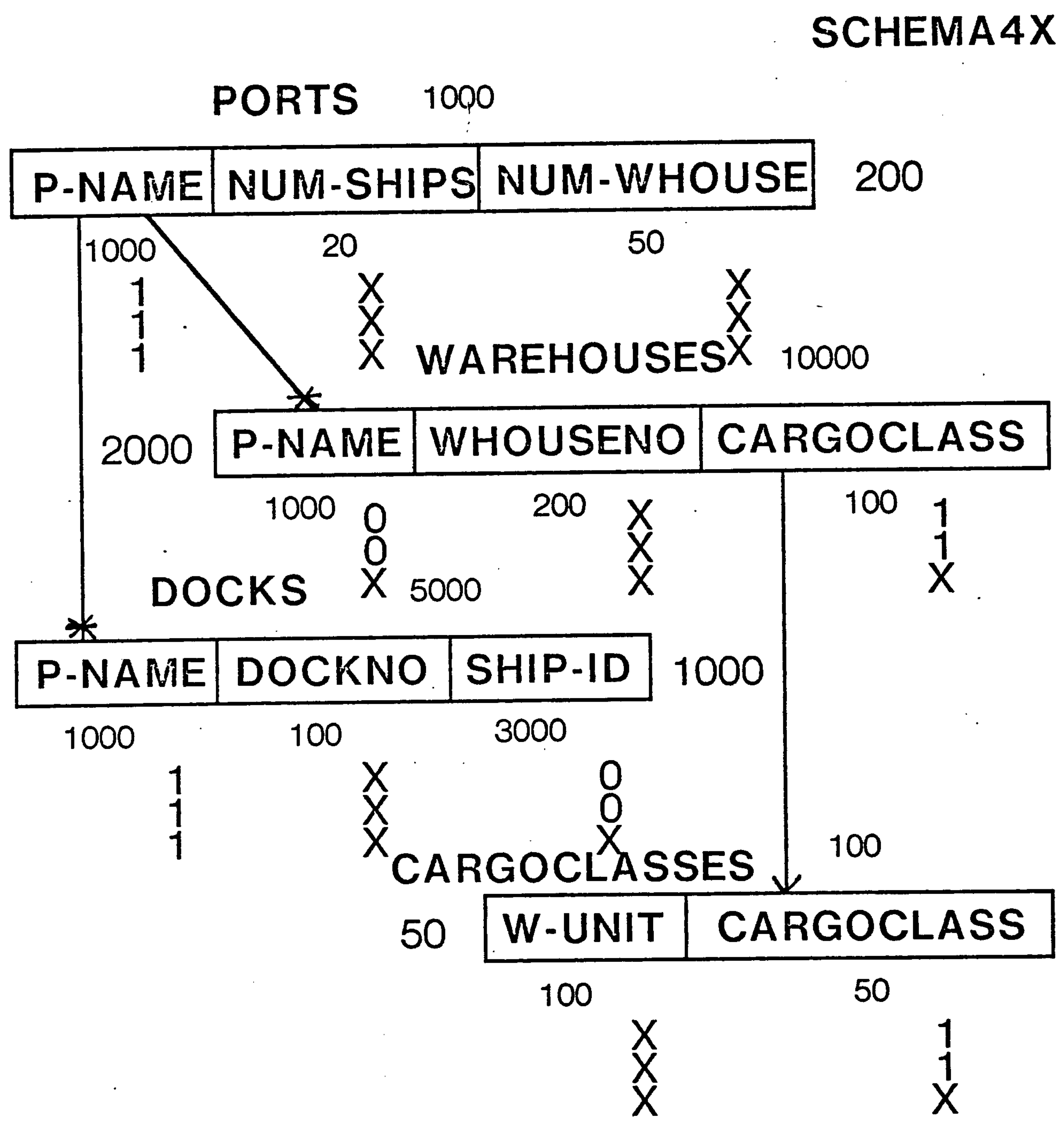


Figure K-5: Situations 40, 41, 42, and their optimal solutions.

```
SCHEMA
    RELATIONS
        RELATION        R1
            RELCARD     200
            NBLOCKS     40
            BLKFAC      5

            COLUMN      C1
                COLCARD         200
                NIBLK           4
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         170
                NIBLK           4
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

        RELATION        R2
            RELCARD     10000
            NBLOCKS     2000
            BLKFAC      5

            COLUMN      C1
                COLCARD         10000
                NIBLK           200
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         200
                NIBLK           200
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C3
                COLCARD         300
                NIBLK           200
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C4
                COLCARD         60
                NIBLK           200
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

        RELATION        R3
            RELCARD     300
            NBLOCKS     60
            BLKFAC      5

            COLUMN      C1
                COLCARD         300
                NIBLK           6
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         100
                NIBLK           6
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

        RELATION        R4
            RELCARD     60
            NBLOCKS     12
            BLKFAC      5

            COLUMN      C1
                COLCARD         60
                NIBLK           2
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         60
                NIBLK           2
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

        RELATION        R5
```

```
            RELCARD     200000
            NBLOCKS     40000
            BLKFAC      5

            COLUMN      C1
                COLCARD         10000
                NIBLK           4000
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         100
                NIBLK           4000
                IBLKFAC         50
                CLUSTERED       0
                INDEX           1

CONNECTIONS
    CONNECTION
        REL1    R1
        COL1    C1
        JSEL1   1.0
        RELN    R2
        COLN    C2
        JSELN   1.0

    CONNECTION
        REL1    R3
        COL1    C1
        JSEL1   1.0
        RELN    R2
        COLN    C3
        JSELN   1.0

    CONNECTION
        REL1    R2
        COL1    C1
        JSEL1   1.0
        RELN    R5
        COLN    C1
        JSELN   1.0

    CONNECTION
        REL1    R4
        COL1    C1
        JSEL1   1.0
        RELN    R2
        COLN    C4
        JSELN   1.0

USAGE
    TRANSACTION 1
        TYPE    JQ      FREQ    20      100     Deleted
        SELECT  R1.C2, R2.C2, R2.C3, R2.C4
        FROM    R1, R2
        WHERE   R1.C1 = R2.C2   AND
                R1.C2 = 100     AND
                R2.C3 = "NAME"

    TRANSACTION 2
        TYPE    JQ      FREQ    20      40      Deleted
        SELECT  R2.C1, R2.C4, R3.C1, R3.C2
        FROM    R2, R3
        WHERE   R2.C3 = R3.C1   AND
                R2.C4 = "KOREA" AND
                R3.C2 = 40

    TRANSACTION 3
        TYPE    JQ      FREQ    20      10      Deleted
        SELECT  R2.C1, R2.C4, R4.C2
        FROM    R2, R4
        WHERE   R2.C4 = R4.C1   AND
                R2.C1 = 101     AND
                R2.C2 = "TANKER"AND
                R4.C2 = 10000000

    TRANSACTION 4
        TYPE    JQ      FREQ    20      60      Deleted
        SELECT  R2.C1, R2.C3, R2.C4, R5.C2
        FROM    R2, R5
        WHERE   R2.C1 = R5.C1   AND
                R2.C4 = "USA"   AND
                R2.C3 = "AMERICAN←OIL←CO"

    TRANSACTION 5
        TYPE    JU      FREQ    5       2.5     2.5
        UPDATE  R2
        SET     R2.C3 = "USA"
        FROM    R2, R5
        WHERE   R2.C1 = R5.C1   AND
                R2.C3 = "BRITAIN" AND
            R5.C2 = 101

TRANSACTION 6
    TYPE    JD      FREQ    5       10      10
    DELETE  R4
    FROM    R4, R2
    WHERE   R4.C1 = R2.C4   AND
            R2.C1 = 101     AND
            R2.C2 = "TANKER"

TRANSACTION 7
    TYPE    SD      FREQ    20      0.01    0.01
    DELETE  R5
    WHERE   R5.C2 > 50

TRANSACTION 8
    TYPE    SU      FREQ    20      1       1
    UPDATE  R2
    SET     R2.C2 = "TANKER"
    WHERE   R2.C4 = "USA"

TRANSACTION 9
    TYPE    SD      FREQ    20      2       2
    DELETE  R2
    WHERE   R2.C2 = "TANKER"

TRANSACTION 10
    TYPE    INS     FREQ    20      120     120
    INSERT  INTO    R1:
            <"TANKER", 50>

TRANSACTION 11
    TYPE    SD      FREQ    20      7       7
    DELETE  R3
    WHERE   R3.C2 > 100

TRANSACTION 12
    TYPE    SU      FREQ    20      80      80
    UPDATE  R4
    SET     R4.C1 = "USA"
    WHERE   R4.C2 = 200000000
```

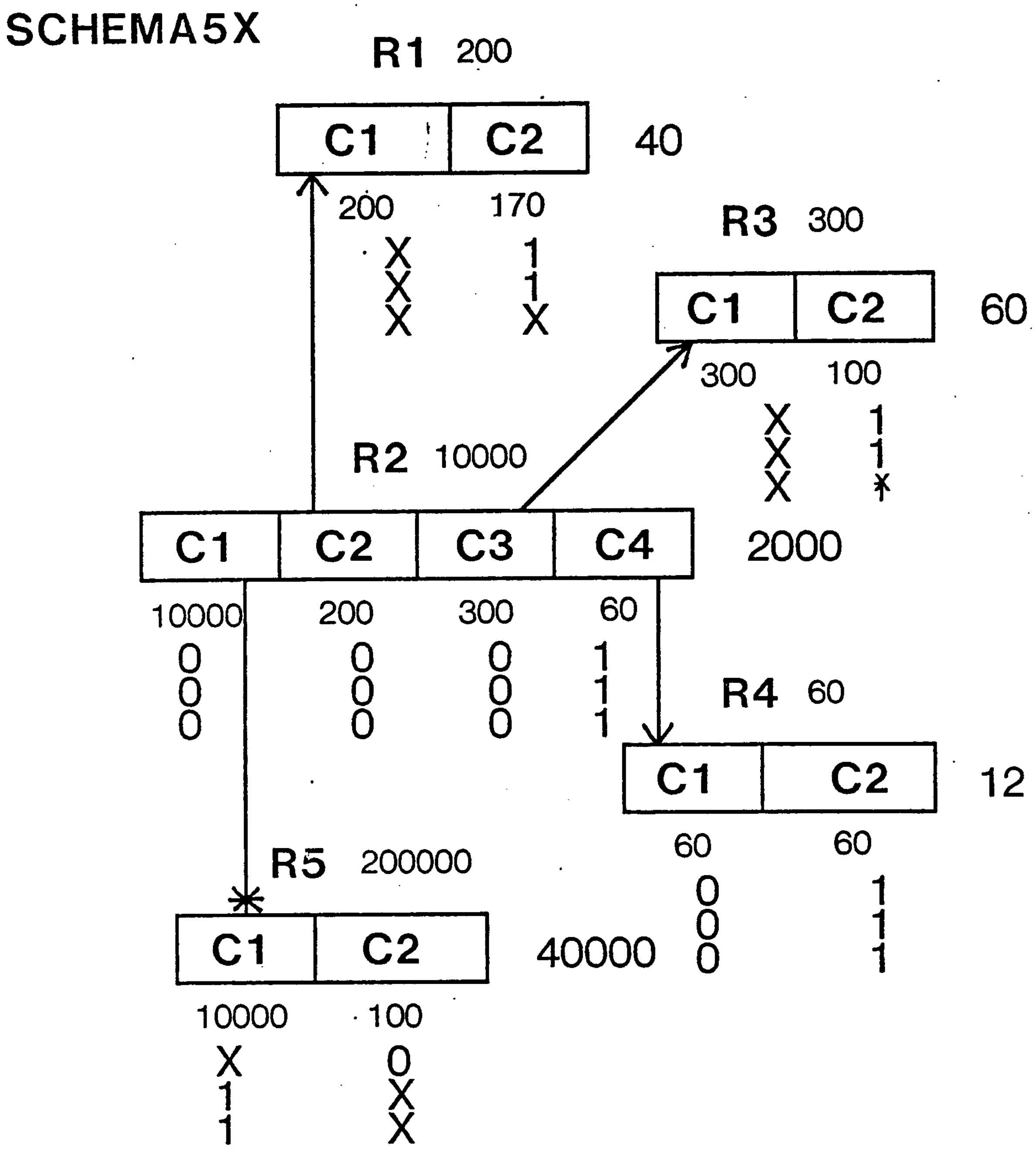


Figure K-6: Situations 50, 51, 52, and their optimal solutions.

```
SCHEMA
    RELATIONS
        RELATION        R1
            RELCARD     1000
            NBLOCKS     200
            BLKFAC      5

            COLUMN      C1
                COLCARD         1000
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         20
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C3
                COLCARD         50 !MAX 50 WAREHOUSES/PORT!
                NIBLK           10
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        R2
            RELCARD     5000
            NBLOCKS     1000
            BLKFAC      5

            COLUMN      C1
                COLCARD         1000
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         100 !DOCK←NUMBER DOES NOT HAVE TO BE NUMBERED
                                     CONTIGUOUSLY!
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C3
                COLCARD         3000 !NOT EVERY DOCK HAS A SHIP ANCHORED!
                NIBLK           50
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        R3
            RELCARD     10000 !10 WAREHOUSED/PORT ON THE AVERAGE!
            NBLOCKS     2000
            BLKFAC      5

            COLUMN      C1
                COLCARD         1000
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C2
                COLCARD         200
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

            COLUMN      C3
                COLCARD         100
                NIBLK           100
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

        RELATION        R4
            RELCARD     100
            NBLOCKS     50
            BLKFAC      2

            COLUMN      C1
                COLCARD         100
                NIBLK           1
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1
            COLUMN      C2
                COLCARD         50
                NIBLK           1
                IBLKFAC         100
                CLUSTERED       0
                INDEX           1

    CONNECTIONS
        CONNECTION
            REL1        R1
            COL1        C1
            JSEL1       1.0
            RELN        R2
            COLN        C1
            JSELN       1.0

        CONNECTION
            REL1        R1
            COL1        C1
            JSEL1       1.0
            RELN        R3
            COLN        C1
            JSELN       1.0

        CONNECTION
            REL1        R4
            COL1        C1
            JSEL1       1.0
            RELN        R3
            COLN        C3
            JSELN       1.0

USAGE
    TRANSACTION 1
        TYPE    JQ      FREQ    100     300     Deleted
        SELECT  R1.C1, R1.C2, R2.C2
        FROM    R1(0.5), R2(0.5)
        WHERE   R1.C1 = R2.C1   AND
                R1.C3 = "ANY"   AND
                R2.C3 = "ANY"   AND
                R1.C2 > "ANY"

    TRANSACTION 2
        TYPE    JQ      FREQ    100     500     Deleted
        SELECT  R1.C3, R2.C1, R2.C3
        FROM    R1(0.5), R2(0.5)
        WHERE   R1.C1 = R2.C1   AND
                R1.C2 = "ANY"   AND
                R2.C2 = "ANY"

    TRANSACTION 3
        TYPE    JQ      FREQ    100     3000    Deleted
        SELECT  R1.C2, R3.C1, R3.C2
        FROM    R1(0.5), R3(0.5)
        WHERE   R1.C1 = R3.C1   AND
                R1.C2 = "ANY"   AND
                R3.C2 = "ANY"   AND
                R3.C3 = "ANY"

    TRANSACTION 4
        TYPE    JQ      FREQ    100     2000    Deleted
        SELECT  R1.C3, R1.C1, R3.C3
        FROM    R1(0.5), R3(0.5)
        WHERE   R1.C1 = R3.C1   AND
                R1.C3 = "ANY"   AND
                R1.C2 = "ANY"   AND
                R3.C2 = "ANY"

    TRANSACTION 5
        TYPE    JQ      FREQ    100     5000    Deleted
        SELECT  R3.C2, R4.C1, R4.C2
        FROM    R3(0.5), R4(0.5)
        WHERE   R3.C3 = R4.C1   AND
                R3.C1 = "ANY"   AND
                R3.C2 = "ANY"   AND
                R4.C2 > "ANY"

    TRANSACTION 6
        TYPE    JQ      FREQ    100     5       Deleted
        SELECT  R3.C1, R3.C3, R4.C2
        FROM    R3(0.5), R4(0.5)
        WHERE   R3.C3 = R4.C1   AND
                R3.C1 > "ANY"

    TRANSACTION 7
        TYPE    AQ      FREQ    100     1000    Deleted
        SELECT  R3.C2, AVG(R3.C3)
        FROM    R3
        WHERE   R3.C1 = "ANY"
        GROUP BY R3.C2
```

```
TRANSACTION 8
    TYPE    SQ      FREQ    100     5000    Deleted
    SELECT  R1.C2
    FROM    R1
    WHERE   R1.C3 = "ANY"   AND
            R1.C1 = "ANY"

TRANSACTION 9
    TYPE    SQ      FREQ    100     5000    Deleted
    SELECT  R2.C3
    FROM    R2
    WHERE   R2.C1 = "ANY"   AND
            R2.C2 = "ANY"

TRANSACTION 10
    TYPE    JU      FREQ    100     100     100
    UPDATE  R3
    SET     R3.C1 = "ANY"
    FROM    R3, R4
    WHERE   R3.C3 = R4.C1   AND
            R3.C2 = "ANY"   AND
            R4.C2 = "ANY"

TRANSACTION 11
    TYPE    JU      FREQ    100     10      10
    UPDATE  R3
    SET     R3.C2 = "ANY"
    FROM    R3, R1
    WHERE   R3.C1 = R1.C1   AND
            R1.C3 = "ANY"

TRANSACTION 12
    TYPE    SD      FREQ    100     100     100
    DELETE  R2
    WHERE   R2.C2 = "ANY"

TRANSACTION 13
    TYPE    SD      FREQ    100     500     500
    DELETE  R1
    WHERE   R1.C3 = "ANY"

TRANSACTION 14
    TYPE    SD      FREQ    100     200     200
    DELETE  R3
    WHERE   R3.C2 = "ANY"

TRANSACTION 15
    TYPE    INS     FREQ    100     5000    5000
    INSERT  INTO    R4:
            <"ANY", "ANY">
```

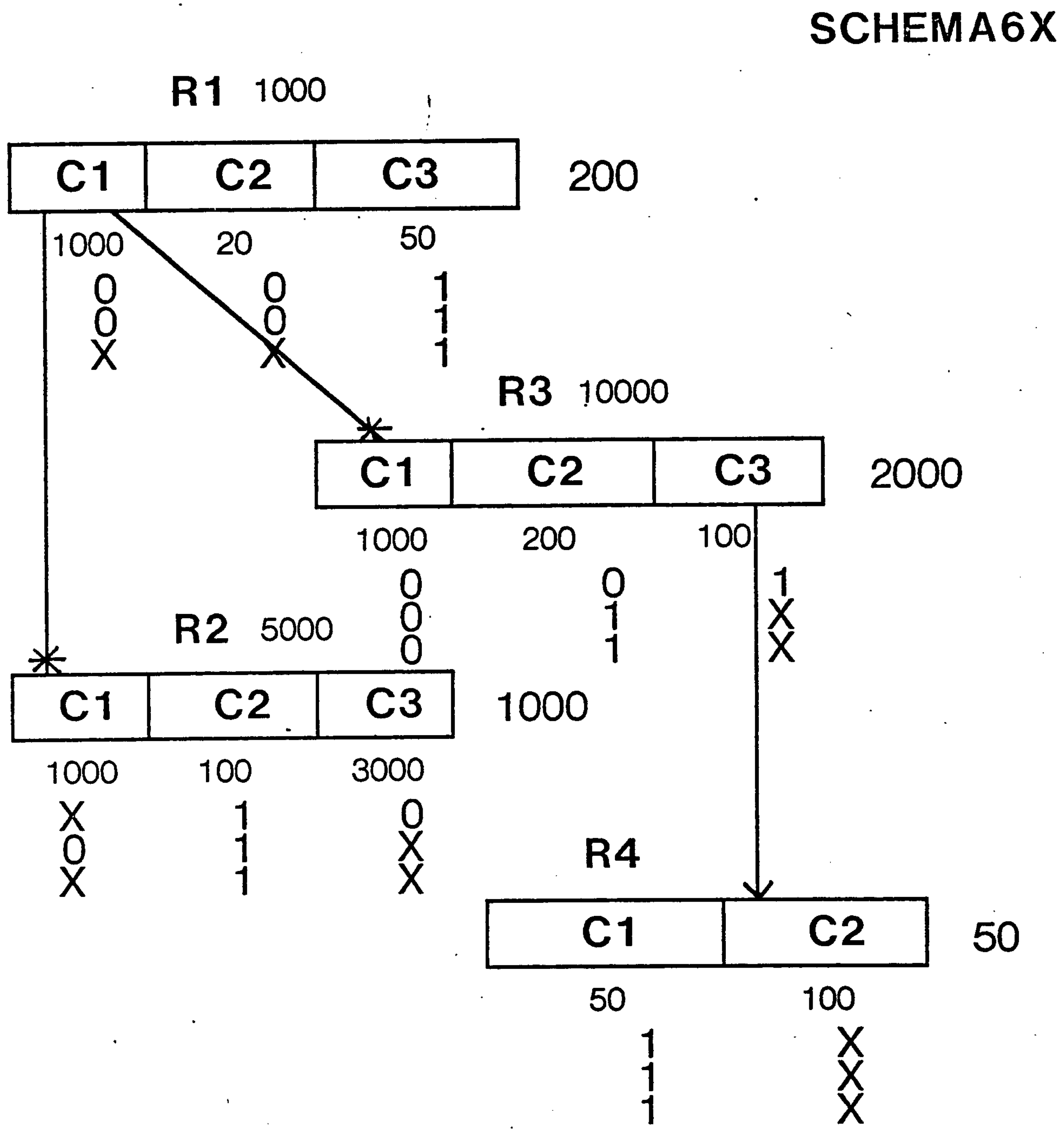


Figure K-7: Situations 60, 61, 62, and their optimal solutions.

```
!512 WORDS = 2560 BYTES/ BLOCK!
SCHEMA
    RELATIONS
        RELATION                FUELTYPES       !15 BYTES!
            RELCARD             8
            NBLOCKS             1
            BLKFAC              173

            COLUMN              FUELTYPE        !5 BYTES!
                COLCARD         8
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1

            COLUMN              PRICE           !5 BYTES!
                COLCARD         8
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1

            COLUMN              UNIT            !5 BYTES!
                COLCARD         4
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1


        RELATION                SHIPTYPES       !1005 BYTES!
            RELCARD             15
            NBLOCKS             8
            BLKFAC              2

            COLUMN              SHIPTYPE        !5 BYTES!
                COLCARD         15
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1

            COLUMN              DESCRIPTION     !1000 BYTES!
                COLCARD         15
                NIBLK           8
                IBLKFAC         2
                CLUSTERED       0
                INDEX           1


        RELATION                SHIPCLASSES     !66 BYTES!
            RELCARD             29
            NBLOCKS             1
            BLKFAC              38

            COLUMN              SHIPTYPE        !5 BYTES!
                COLCARD         15
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1

            COLUMN              SHIPCLASS       !3 BYTES!
                COLCARD         29
                NIBLK           1
                IBLKFAC         320
                CLUSTERED       0
                INDEX           1

            COLUMN              FUELTYPE        !5 BYTES!
                COLCARD         8
                NIBLK           1
                IBLKFAC         256
                CLUSTERED       0
                INDEX           1

            COLUMN              WCAP            !6 BYTES!
                COLCARD         29
                NIBLK           1
                IBLKFAC         284
                CLUSTERED       0
                INDEX           1

            COLUMN              VCAP            !6 BYTES!
                COLCARD         29
                NIBLK           1
                IBLKFAC         284
                CLUSTERED       0
                INDEX           1

            COLUMN              CREWSZ          !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              LIFEBOATCAP     !5 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         256
        CLUSTERED       0
        INDEX           1

    COLUMN              FUELCAP         !5 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         256
        CLUSTERED       0
        INDEX           1

    COLUMN              CRUISESPD       !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXSPD          !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              FUELCONSATMAX   !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              FUELCONSATCRUISING !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              BEAM            !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              LENGTH          !4 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         284
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXDRAFT        !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              DEADWEIGHT      !6 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         284
        CLUSTERED       0
        INDEX           1

RELATION                SHIPS           !98 BYTES!
    RELCARD             2870
    NBLOCKS             111
    BLKFAC              26

    COLUMN              SHIPNAME        !26 BYTES!
        COLCARD         2870
        NIBLK           35
        IBLKFAC         82
        CLUSTERED       0
        INDEX           1

    COLUMN              SHIPID          !5 BYTES!
        COLCARD         2870
        NIBLK           12
        IBLKFAC         256
```

```
            CLUSTERED       0
            INDEX           1

        COLUMN              SHIPCLASS       !3 BYTES!
            COLCARD         29
            NIBLK           9
            IBLKFAC         320
            CLUSTERED       0
            INDEX           1

        COLUMN              IRCS            !6 BYTES!
            COLCARD         2870
            NIBLK           13
            IBLKFAC         232
            CLUSTERED       0
            INDEX           1

        COLUMN              HULLNUMBER      !4 BYTES!
            COLCARD         2870
            NIBLK           11
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              OWNER           !30 BYTES!
            COLCARD         1000
            NIBLK           40
            IBLKFAC         73
            CLUSTERED       0
            INDEX           1

        COLUMN              COUNTRYOFREGISTRY !2 BYTES!
            COLCARD         50
            NIBLK           8
            IBLKFAC         365
            CLUSTERED       0
            INDEX           1

        COLUMN              LATITUDE        !4 BYTES!
            COLCARD         2870
            NIBLK           11
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              NORS            !1 BYTE!
            COLCARD         2
            NIBLK           7
            IBLKFAC         426
            CLUSTERED       0
            INDEX           1

        COLUMN              LONGITUDE       !5 BYTES!
            COLCARD         2870
            NIBLK           11
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              EORW            !1 BYTE!
            COLCARD         2
            NIBLK           7
            IBLKFAC         426
            CLUSTERED       0
            INDEX           1

        COLUMN              DATEREPORTED    !6 BYTES!
            COLCARD         30
            NIBLK           13
            IBLKFAC         232
            CLUSTERED       0
            INDEX           1

        COLUMN              TIMEREPORTED    !4 BYTES!
            COLCARD         975             ! < 24 X 60!
            NIBLK           11
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              ATPORTORSEA     !1 BYTE!
            COLCARD         2
            NIBLK           7
            IBLKFAC         426
            CLUSTERED       0
            INDEX           1


    RELATION                COUNTRIES       !41 BYTES!
        RELCARD             234
        NBLOCKS             4
        BLKFAC              62
    COLUMN              COUNTRYNAME     !30 BYTES!
        COLCARD         234
        NIBLK           4
        IBLKFAC         73
        CLUSTERED       0
        INDEX           1

    COLUMN              COUNTRYABB      !2 BYTES!
        COLCARD         234
        NIBLK           1
        IBLKFAC         365
        CLUSTERED       0
        INDEX           1

    COLUMN              POPULATION      !9 BYTES!
        COLCARD         234
        NIBLK           2
        IBLKFAC         182
        CLUSTERED       0
        INDEX           1


RELATION                SHIPCLASSCARGOCLASS !21 BYTES!
    RELCARD             290             !10 CARGOCLASSES/SHIPCLASS!
    NBLOCKS             3
    BLKFAC              121

    COLUMN              SHIPCLASS       !3 BYTES!
        COLCARD         29
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              CARGOCLASS      !6 BYTES!
        COLCARD         175
        NIBLK           1
        IBLKFAC         232
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXVOLUME       !6 BYTES!
        COLCARD         175
        NIBLK           1
        IBLKFAC         232
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXWEIGHT       !6 BYTES!
        COLCARD         125
        NIBLK           1
        IBLKFAC         232
        CLUSTERED       0
        INDEX           1


RELATION                CARGOCLASSES    !24 BYTES!
    RELCARD             25
    NBLOCKS             1
    BLKFAC              106

    COLUMN              CARGOCLASS      !6 BYTES!
        COLCARD         25
        NIBLK           1
        IBLKFAC         232
        CLUSTERED       0
        INDEX           1

    COLUMN              WUNIT           !9 BYTES!
        COLCARD         20
        NIBLK           1
        IBLKFAC         170
        CLUSTERED       0
        INDEX           1

    COLUMN              VUNIT           !9 BYTES!
        COLCARD         23
        NIBLK           1
        IBLKFAC         170
        CLUSTERED       0
        INDEX           1


RELATION                VOYAGES         !38 BYTES!
    RELCARD             8610            !3 MOST RECENT VOYAGES/SHIP!
    NBLOCKS             129
    BLKFAC              67

    COLUMN              SHIPID          !5 BYTES!
        COLCARD         2870
        NIBLK           34
        IBLKFAC         256
        CLUSTERED       0
        INDEX           1

    COLUMN              VOYAGENUMBER    !3 BYTES!
        COLCARD         350             !MAX 350 VOYAGES/SHIP!
        NIBLK           27
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              CHARTERER       !30 BYTES!
        COLCARD         750
        NIBLK           102
        IBLKFAC         85
        CLUSTERED       0
        INDEX           1

    MCOLUMN             SHIPID←VOYAGENUMBER !8 BYTES!
        COLCARD         8610
        NIBLK           44
        IBLKFAC         196
        COMPONENTS
            SHIPID
            VOYAGENUMBER
        CLUSTERED       0
        INDEX           1


RELATION                LEGS            !17 BYTES!
    RELCARD             17220           !2 LEGS/VOYAGE!
    NBLOCKS             115
    BLKFAC              150

    COLUMN              SHIPID          !5 BYTES!
        COLCARD         2870
        NIBLK           68
        IBLKFAC         256
        CLUSTERED       0
        INDEX           1

    COLUMN              VOYAGENUMBER    !3 BYTES!
        COLCARD         350
        NIBLK           54
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              LEGNUMBER       !3 BYTES!
        COLCARD         10
        NIBLK           54
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              SOURCESTOP      !3 BYTES!
        COLCARD         11              !0..10!
        NIBLK           54
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              DESTINATIONSTOP !3 BYTES!
        COLCARD         11
        NIBLK           54
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    MCOLUMN             SHIPID←VOYAGENUMBER !8 BYTES!
        COLCARD         8610
        NIBLK           88
        IBLKFAC         196
        COMPONENTS
            SHIPID
            VOYAGENUMBER
        CLUSTERED       0
        INDEX           1

    MCOLUMN             SHIPID←VOYAGENUMBER←LEGNUMBER !11 BYTES!
        COLCARD         17220
        NIBLK           108
        IBLKFAC         160
        COMPONENTS
            SHIPID
            VOYAGENUMBER
            LEGNUMBER
        CLUSTERED       0
        INDEX           1

    MCOLUMN             SHIPID←VOYAGENUMBER←SOURCESTOP !11 BYTES!
        COLCARD         17220
```

```
      NIBLK       108
      IBLKFAC     160
      COMPONENTS
         SHIPID
         VOYAGENUMBER
         SOURCESTOP
      CLUSTERED   0
      INDEX       1

   MCOLUMN        SHIPID←VOYAGENUMBER←DESTINATIONSTOP !11 BYTES!
      COLCARD     17220
      NIBLK       108
      IBLKFAC     160
      COMPONENTS
         SHIPID
         VOYAGENUMBER
         DESTINATIONSTOP
      CLUSTERED   0
      INDEX       1


RELATION          STOPS           !52 BYTES!
   RELCARD        25830
   NBLOCKS        528
   BLKFAC         49

   COLUMN         SHIPID          !5 BYTES!
      COLCARD     2870
      NIBLK       101
      IBLKFAC     256
      CLUSTERED   0
      INDEX       1

   COLUMN         VOYAGENUMBER    !3 BYTES!
      COLCARD     350
      NIBLK       81
      IBLKFAC     320
      CLUSTERED   0
      INDEX       1

   COLUMN         STOPNUMBER      !3 BYTES!
      COLCARD     11
      NIBLK       81
      IBLKFAC     320
      CLUSTERED   0
      INDEX       1

   COLUMN         PORTNAME        !18 BYTES!
      COLCARD     100
      NIBLK       233
      IBLKFAC     111
      CLUSTERED   0
      INDEX       1

   COLUMN         ARRIVALDATE     !6 BYTES!
      COLCARD     365             !KEEPS THE RECORD FOR 1 YEAR!
      NIBLK       112
      IBLKFAC     232
      CLUSTERED   0
      INDEX       1

   COLUMN         ARRIVALTIME     !4 BYTES!
      COLCARD     1000            !< 24 X 60!
      NIBLK       91
      IBLKFAC     284
      CLUSTERED   0
      INDEX       1

   COLUMN         DEPARTUREDATE   !6 BYTES!
      COLCARD     365
      NIBLK       112
      IBLKFAC     232
      CLUSTERED   0
      INDEX       1

   COLUMN         DEPARTURETIME   !4 BYTES!
      COLCARD     1000            !< 24 X 60!
      NIBLK       91
      IBLKFAC     284
      CLUSTERED   0
      INDEX       1

   COLUMN         DOCKNUMBER      !3 BYTES!
      COLCARD     15
      NIBLK       81
      IBLKFAC     320
      CLUSTERED   0
      INDEX       1

   MCOLUMN        SHIPID←VOYAGENUMBER !8 BYTES!
      COLCARD     8610
      NIBLK       132
```

```
        IBLKFAC         196
        COMPONENTS
            SHIPID
            VOYAGENUMBER
        CLUSTERED       0
        INDEX           1

    MCOLUMN             SHIPID←VOYAGENUMBER←STOPNUMBER !11 BYTES!
        COLCARD         25830
        NIBLK           162
        IBLKFAC         160
        COMPONENTS
            SHIPID
            VOYAGENUMBER
            STOPNUMBER
        CLUSTERED       0
        INDEX           1

    MCOLUMN             PORTNAME←DOCKNUMBER     !21 BYTES!
        COLCARD         350
        NIBLK           264
        IBLKFAC         98
        COMPONENTS
            PORTNAME
            DOCKNUMBER
        CLUSTERED       0
        INDEX           1


RELATION                DOCKS           !33 BYTES!
    RELCARD             500             !5 DOCKS/PORT!
    NBLOCKS             7
    BLKFAC              77

    COLUMN              PORTNAME        !18 BYTES!
        COLCARD         100
        NIBLK           5
        IBLKFAC         111
        CLUSTERED       0
        INDEX           1

    COLUMN              DOCKNUMBER      !3 BYTES!
        COLCARD         15              !MAX DOCKNUMBER USED!
        NIBLK           2
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              SHIPID          !5 BYTES!
        COLCARD         320
        NIBLK           2
        IBLKFAC         256
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXDRAFT        !3 BYTES!
        COLCARD         200
        NIBLK           2
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXLENGTH       !3 BYTES!
        COLCARD         200
        NIBLK           2
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              OCCUPIEDORNOTOCCUPIED !1 BYTE!
        COLCARD         2
        NIBLK           2
        IBLKFAC         426
        CLUSTERED       0
        INDEX           1

    MCOLUMN             PORTNAME←DOCKNUMBER !21 BYTES!
        COLCARD         500
        NIBLK           5
        IBLKFAC         98
        COMPONENTS
            PORTNAME
            DOCKNUMBER
        CLUSTERED       0
        INDEX           1


RELATION                PORTS           !43 BYTES!
    RELCARD             100
    NBLOCKS             2
    BLKFAC              53

    COLUMN              PORTNAME        !18 BYTES!
        COLCARD         100
        NIBLK           1
        IBLKFAC         111
        CLUSTERED       0
        INDEX           1

    COLUMN              COUNTRY         !2 BYTES!
        COLCARD         70
        NIBLK           1
        IBLKFAC         366
        CLUSTERED       0
        INDEX           1

    COLUMN              LATITUDE        !4 BYTES!
        COLCARD         100
        NIBLK           1
        IBLKFAC         284
        CLUSTERED       0
        INDEX           1

    COLUMN              NORS            !1 BYTE!
        COLCARD         2
        NIBLK           1
        IBLKFAC         426
        CLUSTERED       0
        INDEX           1

    COLUMN              LONGITUDE       !4 BYTES!
        COLCARD         100
        NIBLK           1
        IBLKFAC         284
        CLUSTERED       0
        INDEX           1

    COLUMN              EORW            !1 BYTE!
        COLCARD         2
        NIBLK           1
        IBLKFAC         426
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXDRAFT        !3 BYTES!
        COLCARD         70
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              NUMBEROFDOCKS   !3 BYTES!
        COLCARD         15
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              MAXLENGTH       !3 BYTES!
        COLCARD         70
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1

    COLUMN              NUMBEROFSHIPSATPORT !3 BYTES!
        COLCARD         15
        NIBLK           1
        IBLKFAC         320
        CLUSTERED       0
        INDEX           1


RELATION                WAREHOUSES      !47 BYTES!
    RELCARD             1000            !10 WAREHOUSES/PORT!
    NBLOCKS             19
    BLKFAC              54

    COLUMN              PORTNAME        !18 BYTES!
        COLCARD         100
        NIBLK           10
        IBLKFAC         111
        CLUSTERED       0
        INDEX           1

    COLUMN              WAREHOUSENUMBER !4 BYTES!
        COLCARD         20              !MAX 20 WAREHOUSES/PORT!
        NIBLK           4
        IBLKFAC         284
        CLUSTERED       0
        INDEX           1

    COLUMN              CARGOCLASS      !6 BYTES!
```

```
            COLCARD          25
            NIBLK            5
            IBLKFAC          232
            CLUSTERED        0
            INDEX            1

        COLUMN               USEDORUNUSED     !1 BYTE!
            COLCARD          2
            NIBLK            3
            IBLKFAC          426
            CLUSTERED        0
            INDEX            1

        COLUMN               QUANTITYWEIGHT !6 BYTES!
            COLCARD          100
            NIBLK            5
            IBLKFAC          213
            CLUSTERED        0
            INDEX            1

        COLUMN               QUANTITYVOLUME !6 BYTES!
            COLCARD          100
            NIBLK            5
            IBLKFAC          213
            CLUSTERED        0
            INDEX            1


    RELATION                 LOADEDUNLOADEDCARGOES !30 BYTES!
        RELCARD              77490            !3 CARGOES/STOP!
        NBLOCKS              912
        BLKFAC               85

        COLUMN               SHIPID           !5 BYTES!
            COLCARD          2870
            NIBLK            303
            IBLKFAC          256
            CLUSTERED        0
            INDEX            1

        COLUMN               VOYAGENUMBER     !3 BYTES!
            COLCARD          350
            NIBLK            243
            IBLKFAC          320
            CLUSTERED        0
            INDEX            1

        COLUMN               STOPNUMBER       !3 BYTES!
            COLCARD          11
            NIBLK            243
            IBLKFAC          320
            CLUSTERED        0
            INDEX            1

        COLUMN               CARGOCLASS       !6 BYTES!
            COLCARD          25
            NIBLK            335
            IBLKFAC          232
            CLUSTERED        0
            INDEX            1

        COLUMN               LORU             !1 BYTE!
            COLCARD          2
            NIBLK            182
            IBLKFAC          426
            CLUSTERED        0
            INDEX            1

        COLUMN               QTYWGHT          !6 BYTES!
            COLCARD          10500
            NIBLK            335
            IBLKFAC          232
            CLUSTERED        0
            INDEX            1

        COLUMN               QTYVOL           !6 BYTES!
            COLCARD          9400
            NIBLK            335
            IBLKFAC          232
            CLUSTERED        0
            INDEX            1

        MCOLUMN              SHIPID←VOYAGENUMBER←STOPNUMBER !11 BYTES!
            COLCARD          25830
            NIBLK            485
            IBLKFAC          160
            COMPONENTS
                SHIPID
                VOYAGENUMBER
                STOPNUMBER
            CLUSTERED        0
            INDEX            1
```

```
RELATION                   CARGOESONBOARD  !29 BYTES!
    RELCARD                12000           !5 CARGOES/LEG FOR CURRENT VOYAGE!
    NBLOCKS                137
    BLKFAC                 88

    COLUMN                 SHIPID          !5 BYTES!
        COLCARD            2870
        NIBLK              47
        IBLKFAC            256
        CLUSTERED          0
        INDEX              1

    COLUMN                 VOYAGENUMBER    !3 BYTES!
        COLCARD            350
        NIBLK              38
        IBLKFAC            320
        CLUSTERED          0
        INDEX              1

    COLUMN                 LEGNUMBER       !3 BYTES!
        COLCARD            10
        NIBLK              38
        IBLKFAC            320
        CLUSTERED          0
        INDEX              1

    COLUMN                 CARGOCLASS      !6 BYTES!
        COLCARD            25
        NIBLK              52
        IBLKFAC            232
        CLUSTERED          0
        INDEX              1

    COLUMN                 QUANTITYWEIGHT  !6 BYTES!
        COLCARD            4500
        NIBLK              52
        IBLKFAC            232
        CLUSTERED          0
        INDEX              1

    COLUMN                 QUANTITYVOLUME  !6 BYTES!
        COLCARD            3700
        NIBLK              52
        IBLKFAC            232
        CLUSTERED          0
        INDEX              1

    MCOLUMN                SHIPID←VOYAGENUMBER←LEGNUMBER !11 BYTES!
        COLCARD            2400
        NIBLK              75
        IBLKFAC            160
        COMPONENTS
            SHIPID
            VOYAGENUMBER
            LEGNUMBER
        CLUSTERED          0
        INDEX              1


RELATION                   TRACKS          !68 BYTES!
    RELCARD                4800            !ONLY CURRENT VOYAGE!
    NBLOCKS                65              !AVG 2 REPORT/LEG!
    BLKFAC                 37

    COLUMN                 SHIPID          !5 BYTES!
        COLCARD            1200
        NIBLK              10
        IBLKFAC            256
        CLUSTERED          0
        INDEX              1

    COLUMN                 VOYAGENUMBER    !3 BYTES!
        COLCARD            75
        NIBLK              8
        IBLKFAC            320
        CLUSTERED          0
        INDEX              1

    COLUMN                 LEGNUMBER       !3 BYTES!
        COLCARD            10
        NIBLK              8
        IBLKFAC            320
        CLUSTERED          0
        INDEX              1

    COLUMN                 DATE            !6 BYTES!
        COLCARD            90              !MAX 90 DAYS/LEG!
        NIBLK              11
        IBLKFAC            232
        CLUSTERED          0
```

```
            INDEX           1

        COLUMN              TIME            !4 BYTES!
            COLCARD         1050
            NIBLK           9
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              COURSE          !3 BYTES!
            COLCARD         951
            NIBLK           8
            IBLKFAC         320
            CLUSTERED       0
            INDEX           1

        COLUMN              SPEED           !3 BYTES!
            COLCARD         150
            NIBLK           8
            IBLKFAC         320
            CLUSTERED       0
            INDEX           1

        COLUMN              LATITUDE        !4 BYTES!
            COLCARD         2400
            NIBLK           9
            IBLKFAC         284
            CLUSTERED       0
            INDEX           1

        COLUMN              NORS            !1 BYTE!
            COLCARD         2
            NIBLK           6
            IBLKFAC         426
            CLUSTERED       0
            INDEX           1

        COLUMN              LONGITUDE       !5 BYTES!
            COLCARD         2400
            NIBLK           11
            IBLKFAC         232
            CLUSTERED       0
            INDEX           1

        COLUMN              EORW            !1 BYTE!
            COLCARD         2
            NIBLK           6
            IBLKFAC         426
            CLUSTERED       0
            INDEX           1

        COLUMN              REPORTER        !30 BYTES!
            COLCARD         100
            NIBLK           33
            IBLKFAC         73
            CLUSTERED       0
            INDEX           1

        MCOLUMN             SHIPID←VOYAGENUMBER←LEGNUMBER !11 BYTES!
            COLCARD         2000
            NIBLK           15
            IBLKFAC         160
            COMPONENTS
                SHIPID
                VOYAGENUMBER
                LEGNUMBER
            CLUSTERED       0
            INDEX           1

CONNECTIONS
    CONNECTION                      !1!
        REL1        FUELTYPES
        COL1        FUELTYPE
        JSEL1       1.0
        RELN        SHIPCLASSES
        COLN        FUELTYPE
        JSELN       1.0

    CONNECTION                      !2!
        REL1        SHIPTYPES
        COL1        SHIPTYPE
        JSEL1       1.0
        RELN        SHIPCLASSES
        COLN        SHIPTYPE
        JSELN       1.0

    CONNECTION                      !3!
        REL1        SHIPCLASSES
        COL1        SHIPCLASS
        JSEL1       1.0
        RELN        SHIPCLASSCARGOCLASS
        COLN        SHIPCLASS
```

```
     JSELN           1.0

CONNECTION                             !4!
     REL1            SHIPCLASSES
     COL1            SHIPCLASS
     JSEL1           1.0
     RELN            SHIPS
     COLN            SHIPCLASS
     JSELN           1.0

CONNECTION                             !5!
     REL1            COUNTRIES
     COL1            COUNTRYABB
     JSEL1           0.2137
     RELN            SHIPS
     COLN            COUNTRYOFREGISTRY
     JSELN           1.0

CONNECTION                             !6!
     REL1            COUNTRIES
     COL1            COUNTRYABB
     JSEL1           0.2992
     RELN            PORTS
     COLN            COUNTRY
     JSELN           1.0

CONNECTION                             !7!
     REL1            CARGOCLASSES
     COL1            CARGOCLASS
     JSEL1           1.0
     RELN            SHIPCLASSCARGOCLASS
     COLN            CARGOCLASS
     JSELN           1.0

CONNECTION                             !8!
     REL1            SHIPS
     COL1            SHIPID
     JSEL1           1.0
     RELN            VOYAGES
     COLN            SHIPID
     JSELN           1.0

CONNECTION                             !9!
     REL1            SHIPS
     COL1            SHIPID
     JSEL1           0.1045
     RELN            DOCKS
     COLN            SHIPID
     JSELN           1.0

CONNECTION                             !10!
     REL1            PORTS
     COL1            PORTNAME
     JSEL1           1.0
     RELN            DOCKS
     COLN            PORTNAME
     JSELN           1.0

CONNECTION                             !11!
     REL1            PORTS
     COL1            PORTNAME
     JSEL1           1.0
     RELN            WAREHOUSES
     COLN            PORTNAME
     JSELN           1.0

CONNECTION                             !12!
     REL1            CARGOCLASSES
     COL1            CARGOCLASS
     JSEL1           1.0
     RELN            WAREHOUSES
     COLN            CARGOCLASS
     JSELN           1.0

CONNECTION                             !13!
     REL1            VOYAGES
     COL1            SHIPID←VOYAGENUMBER
     JSEL1           1.0
     RELN            LEGS
     COLN            SHIPID←VOYAGENUMBER
     JSELN           1.0

CONNECTION                             !14!
     REL1            VOYAGES
     COL1            SHIPID←VOYAGENUMBER
     JSEL1           1.0
     RELN            STOPS
     COLN            SHIPID←VOYAGENUMBER
     JSELN           1.0

CONNECTION                             !15!
     REL1            DOCKS
     COL1            PORTNAME←DOCKNUMBER
```

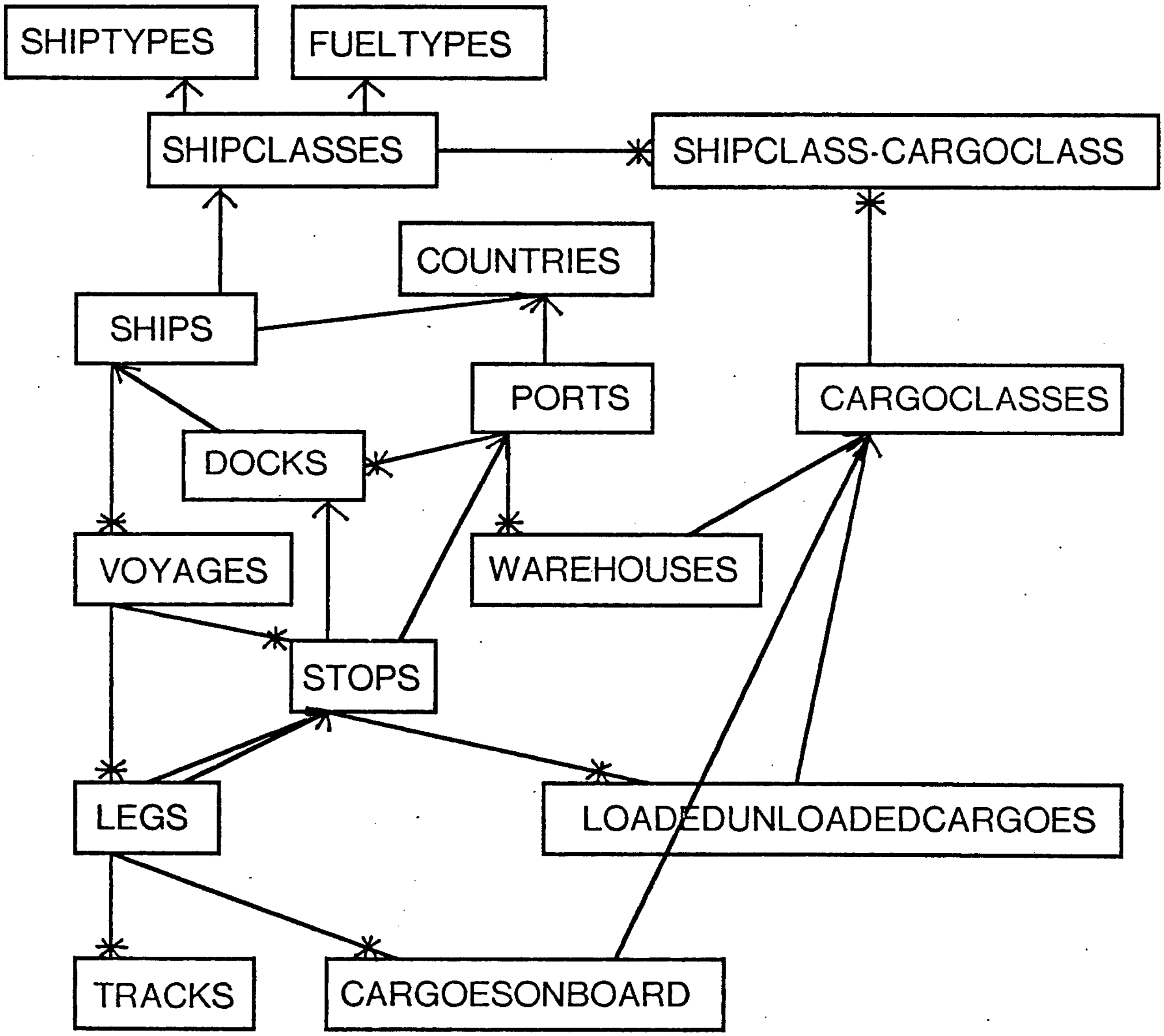
```
            JSEL1       0.7
            RELN        STOPS
            COLN        PORTNAME←DOCKNUMBER
            JSELN       1.0

        CONNECTION                      !16!
            REL1        PORTS
            COL1        PORTNAME
            JSEL1       1.0
            RELN        STOPS
            COLN        PORTNAME
            JSELN       1.0

        CONNECTION                      !17!
            REL1        CARGOCLASSES
            COL1        CARGOCLASS
            JSEL1       1.0
            RELN        CARGOESONBOARD
            COLN        CARGOCLASS
            JSELN       1.0

        CONNECTION                      !18!
            REL1        CARGOCLASSES
            COL1        CARGOCLASS
            JSEL1       1.0
            RELN        LOADEDUNLOADEDCARGOES
            COLN        CARGOCLASS
            JSELN       1.0

        CONNECTION                      !19!
            REL1        STOPS
            COL1        SHIPID←VOYAGENUMBER←STOPNUMBER
            JSEL1       0.6667
            RELN        LEGS
            COLN        SHIPID←VOYAGENUMBER←SOURCESTOP
            JSELN       1.0

        CONNECTION                      !20!
            REL1        STOPS
            COL1        SHIPID←VOYAGENUMBER←STOPNUMBER
            JSEL1       0.6667
            RELN        LEGS
            COLN        SHIPID←VOYAGENUMBER←DESTINATIONSTOP
            JSELN       1.0

        CONNECTION                      !21!
            REL1        STOPS
            COL1        SHIPID←VOYAGENUMBER←STOPNUMBER
            JSEL1       1.0
            RELN        LOADEDUNLOADEDCARGOES
            COLN        SHIPID←VOYAGENUMBER←STOPNUMBER
            JSELN       1.0

        CONNECTION                      !22!
            REL1        LEGS
            COL1        SHIPID←VOYAGENUMBER←LEGNUMBER
            JSEL1       0.2788
            RELN        TRACKS
            COLN        SHIPID←VOYAGENUMBER←LEGNUMBER
            JSELN       1.0

        CONNECTION                      !23!
            REL1        LEGS
            COL1        SHIPID←VOYAGENUMBER←LEGNUMBER
            JSEL1       0.2788
            RELN        CARGOESONBOARD
            COLN        SHIPID←VOYAGENUMBER←LEGNUMBER
            JSELN       1.0


USAGE
    TRANSACTION 1
        TYPE    JQ      FREQ    1000    10000   Deleted
        !SHOW THE PRICE OF THE FUEL FOR THE SHIPTYPE "TIGER"!
        SELECT  SHIPCLASSES.SHIPCLASS, SHIPCLASSES.FUELTYPE, FUELTYPES.PRICE
        FROM    SHIPCLASSES(0.12), FUELTYPES(0.33)
        WHERE   SHIPCLASSES.FUELTYPE = FUELTYPES.FUELTYPE AND
                SHIPCLASSES.SHIPCLASS = "TIGER"

    TRANSACTION 2
        TYPE    JQ      FREQ    1000    10000   Deleted
        !SHOW ALL THE ATTRIBUTES AND DESCRIPTION OF THE SHIPTYPE "LION"!
        SELECT  SHIPCLASSES.*, SHIPTYPES.DESCRIPTION
        FROM    SHIPCLASSES(1), SHIPTYPES(1)
        WHERE   SHIPCLASSES.SHIPTYPE = SHIPTYPES.SHIPTYPE AND
                SHIPCLASSES.SHIPCLASS = "LION"

    TRANSACTION 3
        TYPE    JQ      FREQ    1000    10000   Deleted
        !SHOW SHIPCLASSES THAT CAN CARRY MORE THAN 1000 M3'S LUMBER AND
         THEIR TYPES, VOLUME CAPACITIES AND WEIGHT CAPACITIES!
    SELECT  SHIPCLASSES.SHIPCLASS, SHIPCLASSES.SHIPTYPE,
            SHIPCLASSCARGOCLASS.CARGOCLASS,SHIPCLASSCARGOCLASS.MAXVOLUME,
            SHIPCLASSCARGOCLASS.MAXWEIGHT
    FROM    SHIPCLASSES(0.12), SHIPCLASSCARGOCLASS(1)
    WHERE   SHIPCLASSES.SHIPCLASS = SHIPCLASSCARGOCLASS.SHIPCLASS AND
            SHIPCLASSCARGOCLASS.CARGOCLASS = "LUMBER" AND
            SHIPCLASSCARGOCLASS.MAXVOLUME > 1000

TRANSACTION 4
    TYPE    JQ       FREQ    100      100     Deleted
    !FIND ALL THE TANKERS IN REGION X, THEIR POSITIONS AND COUNTRIES OF
     REGISTRY!
    SELECT  SHIPCLASSES.SHIPTYPE, SHIPS.*
    FROM    SHIPCLASSES(0.12), SHIPS(1)
    WHERE   SHIPCLASSES.SHIPCLASS = SHIPS.SHIPCLASS AND
            SHIPCLASSES.SHIPTYPE = "TANKER" AND
            SHIPS.LATITUDE > 10.0  AND
            SHIPS.LATITUDE < 20.0  AND
            SHIPS.NORS = "N"       AND
            SHIPS.LONGITUDE > 40.0 AND
            SHIPS.LONGITUDE < 60.0 AND
            SHIPS.EORW = "W"

TRANSACTION 5
    TYPE    JQ       FREQ    1000     10000   Deleted
    !SHOW THE COUNTRY OF REGISTRY OF "PACIFIC←PRINCESS"!
    SELECT  SHIPS.SHIPNAME, COUNTRIES.COUNTRYNAME
    FROM    SHIPS(0.29), COUNTRIES(0.78)
    WHERE   SHIPS.COUNTRYOFREGISTRY = COUNTRIES.COUNTRYABB AND
            SHIPS.SHIPNAME = "PACIFIC←PRINCESS"

TRANSACTION 6
    TYPE    JQ       FREQ    200      20000   Deleted
    !SHOW ALL THE ATTRIBUTES AND COUNTRYNAME OF PORT "SANFRANCISCO"!
    SELECT  PORTS.*, COUNTRIES.COUNTRYNAME
    FROM    PORTS(1), COUNTRIES(0.78)
    WHERE   PORTS.COUNTRY = COUNTRIES.COUNTRYABB AND
            PORTS.PORTNAME = "SANFRANCISCO"

TRANSACTION 7
    TYPE    JQ       FREQ    100      10000   Deleted
    !SHOW THE SHIPCLASSES THAT CAN CARRY "LUMBER", THEIR WEIGHT, VOLUME
     CAPACITIES FOR "LUMBER" AND WUNIT, VUNIT OF "LUMBER"!
    SELECT  CARGOCLASSES.CARGOCLASS, CARGOCLASSES.WUNIT,CARGOCLASSES.VUNIT,
            SHIPCLASSCARGOCLASS.SHIPCLASS, SHIPCLASSCARGOCLASS.MAXVOLUME,
            SHIPCLASSCARGOCLASS.MAXWEIGHT
    FROM    CARGOCLASSES(1), SHIPCLASSCARGOCLASS(1)
    WHERE   CARGOCLASSES.CARGOCLASS = SHIPCLASSCARGOCLASS.CARGOCLASS AND
            SHIPCLASSCARGOCLASS.CARGOCLASS = "LUMBER"

TRANSACTION 8
    TYPE    JQ       FREQ    1000     1000    Deleted
    !SHOW THE INFORMATION ABOUT ALL THE SHIPS CHARTERED BY
     "ATLANTIC←OIL←CO"!
    SELECT  VOYAGES.CHARTERER, SHIPS.SHIPNAME, VOYAGES.VOYAGENUMBER,
            SHIPS.IRCS, SHIPS.COUNTRYOFREGISTRY
    FROM    VOYAGES(1), SHIPS(0.40)
    WHERE   VOYAGES.SHIPID = SHIPS.SHIPID  AND
            VOYAGES.CHARTERER = "ATLANTIC←OIL←CO"

TRANSACTION 9
    TYPE    JQ       FREQ    1000     10000   Deleted
    !SHOW THE NAME OF THE SHIP ANCHORED AT SANFRANCISCO DOCK # 7!
    SELECT  DOCKS.PORTNAME, DOCKS.DOCKNUMBER, SHIPS.SHIPNAME
    FROM    DOCKS(0.79), SHIPS(0.32)
    WHERE   DOCKS.SHIPID = SHIPS.SHIPID  AND
            DOCKS.PORTNAME←DOCKNUMBER = "SANFRANCISCO" 7

TRANSACTION 10
    TYPE    JQ       FREQ    100      10000   Deleted
    !SHOW ALL THE ATTRIBUTES OF SANFRANCISCO PORT AND ITS DOCKS!
    SELECT  PORTS.*, DOCKS.*
    FROM    PORTS(1), DOCKS(1)
    WHERE   PORTS.PORTNAME = DOCKS.PORTNAME  AND
            PORTS.PORTNAME = "SANFRANCISCO"

TRANSACTION 11
    TYPE    JQ       FREQ    500      5000    Deleted
    !SHOW THE NAMES OF THE PORTS IN CANADA THAT CAN STORE "EXPLOSIVES"!
    SELECT  PORTS.PORTNAME
    FROM    PORTS(0.42), WAREHOUSES(0.38)
    WHERE   PORTS.PORTNAME = WAREHOUSES.PORTNAME  AND
            PORTS.COUNTRY = "CANADA"  AND
            WAREHOUSES.CARGOCLASS = "EXPLOSIVES"

TRANSACTION 12
    TYPE    JQ       FREQ    200      2000    Deleted
    !SHOW THE ATTRIBUTES OF ALL THE WAREHOUSES OF PORT "SANFRANCISCO" AND
     THE WEIGHT AND VOLUME UNITS OF CARGOCLASSES THEY CAN STORE!
    SELECT  WAREHOUSES.*, CARGOCLASSES.WUNIT, CARGOCLASSES.VUNIT
    FROM    WAREHOUSES(1), CARGOCLASSES(1)
    WHERE   WAREHOUSES.CARGOCLASS = CARGOCLASSES.CARGOCLASS AND
            WAREHOUSES.PORTNAME = "SANFRANCISCO"
```

```
TRANSACTION 13
    TYPE    JQ       FREQ    10000    1000    Deleted
    !SHOW THE RECENT VOYAGES OF SHIP 10105, THEIR LEGS AND CHARTERERS!
    SELECT  VOYAGES.SHIPID, VOYAGES.VOYAGENUMBER,VOYAGES.CHARTERER,
            LEGS.SOURCESTOP, LEGS.DESTINATIONSTOP
    FROM    VOYAGES(1), LEGS(1)
    WHERE   VOYAGES.SHIPID←VOYAGENUMBER = LEGS.SHIPID←VOYAGENUMBER AND
            VOYAGES.SHIPID = 10105

TRANSACTION 14
    TYPE    JQ       FREQ    10000    1000    Deleted
    !SHOW THE RECENT VOYAGES OF SHIP 10105 AND THEIR STOPS!
    SELECT  VOYAGES.SHIPID, VOYAGES.VOYAGENUMBER, VOYAGES.CHARTERER,
            STOPS.*
    FROM    VOYAGES(1), STOPS(1)
    WHERE   VOYAGES.SHIPID←VOYAGENUMBER = STOPS.SHIPID←VOYAGENUMBER AND
            VOYAGES.SHIPID = 10105

TRANSACTION 15
    TYPE    JQ       FREQ    500      5000    Deleted
    !SHOW THE ATTRIBUTES OF THE DOCK AT WHICH SHIP 10105 WILL BE ANCHORED
     AT THE SECOND STOP ON VOYAGE 51!
    SELECT  DOCKS.*
    FROM    STOPS(0.40), DOCKS(1)
    WHERE   STOPS.PORTNAME←DOCKNUMBER = DOCKS.PORTNAME←DOCKNUMBER AND
            STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER = 10105 51 2

TRANSACTION 16
    TYPE    JQ       FREQ    10000    10000   Deleted
    !SHOW THE NAME OF THE PORT AND ITS COUNTRYNAME AT WHICH SHIP 10105
     WILL BE ANCHORED AT THE SECOND STOP ON VOYAGE 51!
    SELECT  PORTS.PORTNAME, PORTS.COUNTRY
    FROM    STOPS(0.35), PORTS(0.47)
    WHERE   STOPS.PORTNAME = PORTS.PORTNAME  AND
            STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER = 10105 51 2

TRANSACTION 17
    TYPE    JQ       FREQ    1000     10000   Deleted
    !SHOW THE CARGOES ON BOARD OF SHIP 10105 AND THEIR WEIGHT, VOLUME, AND
     UNITS ON LEG 2 OF VOYAGE 51!
    SELECT  CARGOESONBOARD.*, CARGOCLASSES.WUNIT, CARGOCLASSES.VUNIT
    FROM    CARGOESONBOARD(1), CARGOCLASSES(1)
    WHERE   CARGOESONBOARD.CARGOCLASS = CARGOCLASSES.CARGOCLASS AND
            CARGOESONBOARD.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 51 2

TRANSACTION 18
    TYPE    JQ       FREQ    1000     10000   Deleted
    !SHOWS THE CARGOES, THEIR WEIGHTS, VOLUMES, AND UNITS THAT SHIP 10105
     UNLOADED AT THE SECOND STOP ON VOYAGE 51!
    SELECT  LOADEDUNLOADEDCARGOES.*, CARGOCLASSES.WUNIT, CARGOCLASSES.VUNIT
    FROM    LOADEDUNLOADEDCARGOES(1), CARGOCLASSES(1)
    WHERE   LOADEDUNLOADEDCARGOES.CARGOCLASS = CARGOCLASSES.CARGOCLASS AND
            LOADEDUNLOADEDCARGOES.SHIPID←VOYAGENUMBER←STOPNUMBER =
                                                      10105 51 2 AND
            LOADEDUNLOADEDCARGOES.LORU = "L"

TRANSACTION 19
    TYPE    JQ       FREQ    100000   10000   Deleted
    !SHOW THE SOURCE STOP'S PORTNAME OF LEG 2 OF VOYAGE 51 OF SHIP 10105!
    SELECT  LEGS.SHIPID, LEGS.VOYAGENUMBER, LEGS.LEGNUMBER, STOPS.PORTNAME
    FROM    LEGS(0.82), STOPS(0.56)
    WHERE   LEGS.SHIPID←VOYAGENUMBER←SOURCESTOP =
                        STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER AND
            LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 51 2

TRANSACTION 20
    TYPE    JQ       FREQ    100000   10000   Deleted
    !SHOW THE DESTINATION STOP'S PORTNAME OF LEG 2 OF VOYAGE 51 OF
     SHIP 10105!
    SELECT  LEGS.SHIPID, LEGS.VOYAGENUMBER, LEGS.LEGNUMBER, STOPS.PORTNAME
    FROM    LEGS(0.82), STOPS(0.56)
    WHERE   LEGS.SHIPID←VOYAGENUMBER←DESTINATIONSTOP =
                        STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER AND
            LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 51 2

TRANSACTION 21
    TYPE    JQ       FREQ    200000   2000    Deleted
    !SHOW ALL THE CARGOES SHIP 10105 LOADED/UNLOADED AT EACH STOP ON
     VOYAGE 51!
    SELECT  STOPS.SHIPID, STOPS.VOYAGENUMBER, STOPS.STOPNUMBER,
            STOPS.PORTNAME, LOADEDUNLOADEDCARGOES.*
    FROM    STOPS(0.56), LOADEDUNLOADEDCARGOES(1)
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER =
                LOADEDUNLOADEDCARGOES.SHIPID←VOYAGENUMBER←STOPNUMBER  AND
            STOPS.SHIPID←VOYAGENUMBER = 10105 51

TRANSACTION 22
    TYPE    JQ       FREQ    5000     5000    Deleted
    !SHOW THE LEGS AND THEIR TRACK INFORMATION OF SHIP 10105!
    SELECT  LEGS.SHIPID, LEGS.SOURCESTOP, LEGS.DESTINATIONSTOP, TRACKS.*
    FROM    LEGS(1), TRACKS(1)
    WHERE   LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER =
                TRACKS.SHIPID←VOYAGENUMBER←LEGNUMBER AND
        LEGS.SHIPID = 10105

TRANSACTION 23
    TYPE    JQ      FREQ    100000  10000   Deleted
    !SHOW THE SOURCE STOP, DESTINATION STOP OF LEG 2 OF VOYAGE 51 OF
     SHIP 10105 AND CARGOES ON BOARD ON THAT LEG!
    SELECT  LEGS.SOURCESTOP, LEGS.DESTINATIONSTOP, CARGOESONBOARD.*
    FROM    LEGS(1), CARGOESONBOARD(1)
    WHERE   LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER =
                CARGOESONBOARD.SHIPID←VOYAGENUMBER←LEGNUMBER AND
            LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 51 2

TRANSACTION 104
    TYPE    JQ      FREQ    200     200     Deleted
    !SHOW THE NAME, TYPE, AND DEADWEIGHT OF SHIPS REGISTERED IN NETHERLANDS!
    SELECT  SHIPS.COUNTRYOFREGISTRY, SHIPS.SHIPNAME,
            SHIPCLASSES.SHIPTYPE, SHIPCLASSES.DEADWEIGHT
    FROM    SHIPS(0.32), SHIPCLASSES(0.21)
    WHERE   SHIPS.SHIPCLASS = SHIPCLASSES.SHIPCLASS AND
            SHIPS.COUNTRYOFREGISTRY = "NT"

TRANSACTION 108
    TYPE    JQ      FREQ    200     2000    Deleted
    !SHOW ALL THE SHIPS OWNED BY "ONASIS" AND THEIR VOYAGES AND CHARTERERS!
    SELECT  SHIPS.OWNER, SHIPS.SHIPNAME,
            VOYAGES.VOYAGENUMBER, VOYAGES.CHARTERER
    FROM    SHIPS(0.62), VOYAGES(1)
    WHERE   SHIPS.SHIPID = VOYAGES.SHIPID AND
            SHIPS.OWNER = "ONASIS"

TRANSACTION 110
    TYPE    JQ      FREQ    100     1000    Deleted
    !FIND THE PORTS AND THEIR LOCATIONS THAT HAVE DOCKS MORE THAN 50 FT
     DEEP!
    SELECT  PORTS.PORTNAME, PORTS.COUNTRY, PORTS.LATITUDE, PORTS.NORS,
            PORTS.LATITUDE, PORTS.EORW, DOCKS.DOCKNUMBER, DOCKS.MAXLENGTH
    FROM    PORTS(0.70), DOCKS(0.73)
    WHERE   PORTS.PORTNAME = DOCKS.PORTNAME AND
            DOCKS.MAXDRAFT > 50

TRANSACTION 113
    TYPE    JQ      FREQ    1000    1000    Deleted
    !SHOW ALL THE VOYAGES AND THEIR LEGS THAT ARE CHARTERED BY
     "ATLANTIC←OIL←CO"!
    SELECT  VOYAGES.SHIPID, VOYAGES.VOYAGENUMBER, LEGS.*
    FROM    VOYAGES(1), LEGS(1)
    WHERE   VOYAGES.SHIPID←VOYAGENUMBER = LEGS.SHIPID←VOYAGENUMBER AND
            VOYAGES.CHARTERER = "ATLANTIC←OIL←CO"

TRANSACTION 121
    TYPE    JQ      FREQ    100     10000   Deleted
    !FIND THE PORTS WHERE SHIP 10105 UNLOADED "LUMBER" ON VOYAGE 51,
     AND THE TIME THE SHIP ARRIVED AT THESE PORTS!
    SELECT  STOPS.PORTNAME, STOPS.ARRIVALDATE, STOPS.ARRIVALTIME
    FROM    STOPS(0.75), LOADEDUNLOADEDCARGOES(0.37)
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER =
               LOADEDUNLOADEDCARGOES.SHIPID←VOYAGENUMBER←STOPNUMBER AND
            LOADEDUNLOADEDCARGOES.SHIPID = 10105 AND
            LOADEDUNLOADEDCARGOES.VOYAGENUMBER = 51 AND
            LOADEDUNLOADEDCARGOES.CARGOCLASS = "LUMBER" AND
            LOADEDUNLOADEDCARGOES.LORU = "U"

TRANSACTION 122
    TYPE    JQ      FREQ    500     5000    Deleted
    !FIND THE DESTINATION, COURSE AND SPEED OF THE SHIP TRACKED BY
     THE SITE AT "PORTSMOUTH" AT 18:22 ON JUNE 22, 1982!
    SELECT  LEGS.DESTINATIONSTOP, TRACKS.COURSE, TRACKS.SPEED
    FROM    TRACKS(0.25), LEGS(0.82)
    WHERE   LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER =
                TRACKS.SHIPID←VOYAGENUMBER←LEGNUMBER AND
            TRACKS.DATE = 062682 AND
            TRACKS.TIME = 1822   AND
            TRACKS.REPORTER = "PORTSMOUTH"

TRANSACTION 201
    TYPE    SQ      FREQ    100     10000   Deleted
    !FIND SHIPCLASSES OF TYPE "TRAWLER" AND THEIR DEADWEIGHT AND
     CRUISING SPEED!
    SELECT  SHIPCLASSES.SHIPCLASS, SHIPCLASSES.DEADWEIGHT,
            SHIPCLASSES.CRUISESPD
    FROM    SHIPCLASSES
    WHERE   SHIPCLASSES.SHIPTYPE = "TRAWLER"

TRANSACTION 202
    TYPE    SQ      FREQ    200     20000   Deleted
    !FIND SHIPCLASSES AND THEIR TYPES WHOSE DEADWEIGHTS EXCEED 10000 TONS!
    SELECT  SHIPCLASSES.SHIPCLASS, SHIPCLASSES.SHIPTYPES
    FROM    SHIPCLASSES
    WHERE   SHIPCLASSES.DEADWEIGHT > 10000

TRANSACTION 203
    TYPE    SQ      FREQ    5000    5000    Deleted
```

```
        !FIND THE IRCS AND THE POSITION OF "QE2"!
        SELECT  SHIPS.IRCS, SHIPS.LONGITUDE, SHIPS.EORW, SHIPS.LATITUDE,
                SHIPS.NORS
        FROM    SHIPS
        WHERE   SHIPS.SHIPNAME = "QE2"

    TRANSACTION 204
        TYPE    SQ       FREQ    100      10000   Deleted
        !SHOW ALL THE ATTRIBUTES OF PORT "ALEXANDRIA"!
        SELECT  PORTS.*
        FROM    PORTS
        WHERE   PORTS.PORTNAME = "ALEXANDRIA"

    TRANSACTION 205
        TYPE    SQ       FREQ    100      10000   Deleted
        !FIND ALL THE PORTS IN "FRANCE"!
        SELECT  PORTS.PORTNAME
        FROM    PORTS
        WHERE   PORTS.COUNTRY = "FRANCE"

    TRANSACTION 206
        TYPE    SQ       FREQ    100      10000   Deleted
        !DESCRIBE ALL THE ATTRIBUTES OF DOCKS IN MARSEILLES!
        SELECT  DOCKS.*
        FROM    DOCKS
        WHERE   DOCKS.PORTNAME = "MARSEILLES"

    TRANSACTION 207
        TYPE    SQ       FREQ    5000     5000    Deleted
        !SHOW THE CARGOES, VOLUME CAPACITY AND WEIGHT CAPACITY OF WAREHOUSE 5
         OF PORT MARSEILLES!
        SELECT  WAREHOUSES.CARGOCLASS, WAREHOUSES.QUANTITYWEIGHT,
                WAREHOUSES.QUANTITYVOLUME
        FROM    WAREHOUSES
        WHERE   WAREHOUSES.PORTNAME = "MARSEILLES" AND
                WAREHOUSES.WAREHOUSENUMBER = 5

    TRANSACTION 208
        TYPE    SQ       FREQ    40000    20000   Deleted
        !SHOW ALL THE ATTRIBUTES OF THE STOPS THE SHIP 10105 MADE ON VOYAGE 51!
        SELECT  *
        FROM    STOPS
        WHERE   STOPS.SHIPID = 10105 AND
                STOPS.VOYAGENUMBER = 51

    TRANSACTION 209
        TYPE    SQ       FREQ    24000    24000   Deleted
        !FIND THE CHARTERER OF VOYAGE 51 OF THE SHIP 10105!
        SELECT  VOYAGE.CHARTERER
        FROM    VOYAGES
        WHERE   VOYAGES.SHIPID←VOYAGENUMBER = 10105 51

!FOR TRANSACTIONS OF TYPE AQ, IF AGGREGATION OPERATORS COUNT,AVG,SUM ARE USED,
 THE PROJECTION FACTOR MUST BE 1, SINCE DUPLICATES SHOULD NOT BE REMOVED.
 IF MIN, MAX ARE USED, THE PROJECTION FACTOR DEPENDS ON THE SELECTED FIELDS
 AND THE FIELDS IN THE GROUP BY CLAUSE.!

    TRANSACTION 301
        TYPE    AQ       FREQ    100      100     Deleted
        !SHOW THE OWNERS WHO OWN MORE THAN 10 SHIPS AND HOW MANY SHIPS
         THEY OWN!
        SELECT  SHIPS.OWNER, COUNT(*)
        FROM    SHIPS
        GROUP BY SHIPS.OWNER
        HAVING  COUNT(*) > 10

    TRANSACTION 302
        TYPE    AQ       FREQ    100      1000    Deleted
        !FIND THE AVERAGE MAXDRAFT OVER ALL DOCKS OF EACH PORT!
        SELECT  DOCKS.PORTNAME, AVG(DOCKS.MAXDRAFT)
        FROM    DOCKS
        GROUP BY DOCKS.PORTNAME

    TRANSACTION 303
        TYPE    AQ       FREQ    100      100     Deleted
        !SHOW HOW MANY VOYAGES EACH CHARTERER CHARTERED FOR 1 YEAR!
        SELECT  VOYAGES.CHARTERER, COUNT(*)
        FROM    VOYAGES
        GROUP BY VOYAGES.CHARTERER

    TRANSACTION 304
        TYPE    AQ       FREQ    200      20      Deleted
        !SHOW HOW MANY SHIPS USED EACH PORT FROM JAN 1, 1982 TO JUNE 30, 1982!
        SELECT  STOPS.PORTNAME, COUNT(*)
        FROM    STOPS
        WHERE   STOPS.ARRIVALDATE > 010181 AND
                STOPS.ARRIVALDATE < 063082
        GROUP BY STOPS.PORTNAME

    TRANSACTION 305
        TYPE    AQ       FREQ    1000     10000   Deleted
        !SHOW THE TOTAL WEIGHT AND VOLUME OF CARGOES ON BOARD OF SHIP 10105
         DURING THE SECOND LEG OF VOYAGE 51!
    SELECT   SUM(CARGOESONBOARD.QUANTITYWEIGHT),
             SUM(CARGOESONBOARD.QUANTITYVOLUME)
    FROM     CARGOESONBOARD
    WHERE    CARGOESONBOARD.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 51 2

TRANSACTION 401
    TYPE     INS      FREQ    0.1      10000    10000
    INSERT   INTO     FUELTYPES:
             <"OIL←C", 30, "GALLON">

TRANSACTION 402
    TYPE     INS      FREQ    0.1      10000    10000
    INSERT   INTO     SHIPTYPES:
             <"TRAWLER", "A←FISHING←VESSEL←WHICH←USES←A←TRAWLNET">

TRANSACTION 403
    TYPE     INS      FREQ    10       10000    10000
    INSERT   INTO     SHIPCLASSES:
             <"TANKER", "SX←7", "OTHER←ATTRIBUTES">

TRANSACTION 404
    TYPE     SD       FREQ    10       10000    10000
    DELETE   SHIPCLASSES
    WHERE    SHIPCLASSES.SHIPCLASS = "DRAGON"

TRANSACTION 405
    TYPE     INS      FREQ    100      10000    10000
    INSERT   INTO     SHIPCLASSCARGOCLASS:
             <"SX←7", "GRAIN", 20000, 10000>

TRANSACTION 406
    TYPE     SD       FREQ    10       5000     5000
    DELETE   SHIPCLASSCARGOCLASS
    WHERE    SHIPCLASSCARGOCLASS.SHIPCLASS = "DRAGON"

TRANSACTION 407
    TYPE     INS      FREQ    0.1      10000    10000
    INSERT   INTO     COUNTRIES:
             <"NC", "NEW←COUNTRY", 1000000>

TRANSACTION 408
    TYPE     INS      FREQ    5        5000     5000
    INSERT   INTO     PORTS:
             <"KUMI", "KR", "OTHER←ATTRIBUTES">

TRANSACTION 409
    TYPE     SD       FREQ    1        10000    10000
    DELETE   PORTS
    WHERE    PORTS.PORTNAME = "OLD←PORT"

TRANSACTION 410
    TYPE     INS      FREQ    10       10000    10000
    INSERT   INTO     CARGOCLASSES:
             <"NEW←CARGOCLASS", "TON", "M3">

TRANSACTION 411
    TYPE     INS      FREQ    50       5000     5000
    INSERT   INTO     DOCKS:
             <0, "KUMI", 1, 50, 200, "N">

TRANSACTION 412
    TYPE     INS      FREQ    500      5000     5000
    INSERT   INTO     WAREHOUSES:
             <"BOSTON", 11, "OTHER←ATTRIBUTES">

TRANSACTION 413
    TYPE     INS      FREQ    8610     4300     4300
    INSERT   INTO     VOYAGES:
             <10105, 320, "ATLANTIC←OIL←CO">

TRANSACTION 414
    TYPE     SD       FREQ    8610     4300     4300
    DELETE   VOYAGES
    WHERE    VOYAGES.SHIPID←VOYAGENUMBER = 10105 320

TRANSACTION 415
    TYPE     INS      FREQ    25830    2583     2583
    INSERT   INTO     STOPS:
             <10105, 320, 5, "OTHER←ATTRIBUTES">

TRANSACTION 416
    TYPE     SD       FREQ    25830    2583     2583
    DELETE   STOPS
    WHERE    STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER = 10105 320 5

TRANSACTION 417
    TYPE     INS      FREQ    77490    7749     7749
    INSERT   INTO     LOADEDUNLOADEDCARGOES:
             <10105, 320, 5, "OTHER←ATTRIBUTES">

TRANSACTION 418
    TYPE     SD       FREQ    77490    7749     7749
    DELETE   LOADEDUNLOADEDCARGOES
    WHERE    LOADEDUNLOADEDCARGOES.SHIPID←VOYAGENUMBER←STOPNUMBER =
                                                10105 320 5

TRANSACTION 419
    TYPE     INS      FREQ     17220    2000     2000
    INSERT   INTO     LEGS:
             <10105, 320, 5, "OTHER←ATTRIBUTES">

TRANSACTION 420
    TYPE     SD       FREQ     17220    2000     2000
    DELETE   LEGS
    WHERE    LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 32 5

TRANSACTION 421
    TYPE     INS      FREQ     12000    12000    12000
    INSERT   INTO     CARGOESONBOARD:
             <10105, 320, 5, "OTHER←ATTRIBUTES">

TRANSACTION 422
    TYPE     SD       FREQ     12000    12000    12000
    DELETE   CARGOESONBOARD
    WHERE    CARGOESONBOARD.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 32 5

TRANSACTION 423
    TYPE     INS      FREQ     2400     2400     2400
    INSERT   INTO     TRACKS:
             <10105, 320, 5, "OTHER←ATTRIBUTES">

TRANSACTION 424
    TYPE     SD       FREQ     2400     2400     2400
    DELETE   TRACKS
    WHERE    TRACKS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 32 5

TRANSACTION 501
    TYPE     SU       FREQ     100      10000    10000
    UPDATE   FUELTYPES
    SET      FUELTYPES.PRICE = 140
    WHERE    FUELTYPES.FUELTYPE = "GASOLINE"

TRANSACTION 502
    TYPE     SU       FREQ     500      500      500
    UPDATE   SHIPS
    SET      SHIPS.OWNER = "PACIFIC←TRADING←CO",
    SET      SHIPS.SHIPNAME = "TRADE←WIND"
    WHERE    SHIPS.SHIPID = 10105

TRANSACTION 503
    TYPE     SU       FREQ     200      2000     2000
    UPDATE   SHIPS
    SET      SHIPS.COUNTRYOFREGISTRY = "SPAIN"
    WHERE    SHIPS.SHIPID = 10105

TRANSACTION 504
    TYPE     SU       FREQ     17220    17220    17220
    UPDATE   SHIPS
    SET      SHIPS.LATITUDE = 20.45,
    SET      SHIPS.NORS = "N",
    SET      SHIPS.LONGITUDE = 40.00,
    SET      SHIPS.EORW = "W",
    SET      SHIPS.DATEREPORTED = 063082,
    SET      SHIPS.TIMEREPORTED = 1724,
    SET      SHIPS.ATPORTORSEA = "S"
    WHERE    SHIPS.SHIPID = 10105

TRANSACTION 505
    TYPE     SU       FREQ     234      11700    11700
    UPDATE   COUNTRIES
    SET      COUNTRIES.POPULATION = 35000000
    WHERE    COUNTRIES.COUNTRYABB = "KR"

TRANSACTION 506
    TYPE     SU       FREQ     10000    10000    10000
    UPDATE   PORTS
    SET      PORTS.NUMBEROFSHIPSATPORT = 15
    WHERE    PORTS.PORTNAME = "NEWORLEANS"

TRANSACTION 507
    TYPE     SU       FREQ     15000    15000    15000
    UPDATE   DOCKS
    SET      DOCKS.SHIPID = 10105,
    SET      DOCKS.OCCUPIEDORNOTOCCUPIED = "O"
    WHERE    DOCKS.PORTNAME←DOCKNUMBER = "NEWORLEANS←5"

TRANSACTION 508
    TYPE     SU       FREQ     2000     20000    20000
    UPDATE   WAREHOUSES
    SET      WAREHOUSES.USEDORUNUSED = "Y"
    WHERE    WAREHOUSES.PORTNAME = "NEWORLEANS" AND
             WAREHOUSES.WAREHOUSENUMBER = 7

TRANSACTION 509
    TYPE     SU       FREQ     25830    8600     8600
    UPDATE   STOPS
    SET     STOPS.ARRIVALDATE = 063082,
    SET     STOPS.ARRIVALTIME = 1545,
    SET     STOPS.DOCKNUMBER = 7
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER = 10105 3203

TRANSACTION 510
    TYPE    SU      FREQ    25830   8600    8600
    UPDATE  STOPS
    SET     STOPS.DEPARTUREDATE = 070582,
    SET     STOPS.DEPARTURETIME = 0542
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER = 10105 320 3

TRANSACTION 511
    TYPE    INS     FREQ    100     200     200
    INSERT  INTO    SHIPS:
            <"NEW←SHIP", 10105, "OTHER←ATTRIBUTES">

TRANSACTION 512
    TYPE    SD      FREQ    50      20      20
    DELETE  SHIPS
    WHERE   SHIPS.SHIPID = 10105

TRANSACTION 601
    TYPE    JU      FREQ    170     170     170
    UPDATE  STOPS
    SET     STOPS.PORTNAME = "LONDON"
    FROM    STOPS, LEGS(0.65)
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER =
                    LEGS.SHIPID←VOYAGENUMBER←DESTINATIONSTOP  AND
            LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 320 4

TRANSACTION 602
    TYPE    JU      FREQ    100     100     100
    UPDATE  STOPS
    SET     STOPS.PORTNAME = "LISBON"
    FROM    STOPS, LEGS(0.65)
    WHERE   STOPS.SHIPID←VOYAGENUMBER←STOPNUMBER =
                    LEGS.SHIPID←VOYAGENUMBER←SOURCESTOP AND
            LEGS.SHIPID←VOYAGENUMBER←LEGNUMBER = 10105 320 5
```

SCHEMA7X KBMS DATABASE

110 ATTRIBUTES IN 16 RELATIONS



```
OPTIMAL ACCESS CONFIGURATION FOR SITUATION 70

TOTALCOST =  5.415462017E+06

RELATION   TRACKS

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

DATE
INDEX =  FALSE  CLUSTERED =  FALSE

TIME
INDEX =   TRUE  CLUSTERED =  FALSE

COURSE
INDEX =  FALSE  CLUSTERED =  FALSE

SPEED
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

REPORTER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   CARGOESONBOARD

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   LOADEDUNLOADEDCARGOES

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

LORU
INDEX =  FALSE  CLUSTERED =  FALSE

QTYWGHT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
QTYVOL
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   WAREHOUSES

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

WAREHOUSENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

USEDORUNUSED
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   PORTS

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

COUNTRY
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFDOCKS
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFSHIPSATPORT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   DOCKS

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

OCCUPIEDORNOTOCCUPIED
INDEX =  FALSE  CLUSTERED =  FALSE

PORTNAME←DOCKNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   STOPS

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE
```

```
VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

PORTNAME
INDEX =  FALSE  CLUSTERED =  FALSE

ARRIVALDATE
INDEX =  FALSE  CLUSTERED =  FALSE

ARRIVALTIME
INDEX =  FALSE  CLUSTERED =  FALSE

DEPARTUREDATE
INDEX =  FALSE  CLUSTERED =  FALSE

DEPARTURETIME
INDEX =  FALSE  CLUSTERED =  FALSE

DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =   TRUE  CLUSTERED =  FALSE

PORTNAME←DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   LEGS

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SOURCESTOP
INDEX =  FALSE  CLUSTERED =  FALSE

DESTINATIONSTOP
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←SOURCESTOP
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←DESTINATIONSTOP
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   VOYAGES

SHIPID
INDEX =   TRUE  CLUSTERED =  FALSE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CHARTERER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =  FALSE


RELATION   CARGOCLASSES

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

WUNIT
INDEX =  FALSE  CLUSTERED =  FALSE

VUNIT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
RELATION   SHIPCLASSCARGOCLASS

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =   TRUE  CLUSTERED =   TRUE

MAXVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

MAXWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   COUNTRIES

COUNTRYNAME
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRYABB
INDEX =   TRUE  CLUSTERED =   TRUE

POPULATION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPS

SHIPNAME
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPID
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

IRCS
INDEX =  FALSE  CLUSTERED =  FALSE

HULLNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

OWNER
INDEX =   TRUE  CLUSTERED =   TRUE

COUNTRYOFREGISTRY
INDEX =   TRUE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

DATEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

TIMEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

ATPORTORSEA
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPCLASSES

SHIPTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

WCAP
INDEX =  FALSE  CLUSTERED =  FALSE

VCAP
```

```
INDEX =  FALSE  CLUSTERED =  FALSE

CREWSZ
INDEX =  FALSE  CLUSTERED =  FALSE

LIFEBOATCAP
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCAP
INDEX =  FALSE  CLUSTERED =  FALSE

CRUISESPD
INDEX =  FALSE  CLUSTERED =  FALSE

MAXSPD
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATMAX
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATCRUISING
INDEX =  FALSE  CLUSTERED =  FALSE

BEAM
INDEX =  FALSE  CLUSTERED =  FALSE

LENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

DEADWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPTYPES

SHIPTYPE
INDEX =    TRUE  CLUSTERED =   TRUE

DESCRIPTION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   FUELTYPES

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

PRICE
INDEX =  FALSE  CLUSTERED =  FALSE

UNIT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
OPTIMAL ACCESS CONFIGURATION FOR SITUATION 71

TOTALCOST =  2.172586202E+06

RELATION   TRACKS

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

DATE
INDEX =  FALSE  CLUSTERED =  FALSE

TIME
INDEX =   TRUE  CLUSTERED =  FALSE

COURSE
INDEX =  FALSE  CLUSTERED =  FALSE

SPEED
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

REPORTER
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   CARGOESONBOARD

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   LOADEDUNLOADEDCARGOES

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

LORU
INDEX =  FALSE  CLUSTERED =  FALSE

QTYWGHT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
QTYVOL
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   WAREHOUSES

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

WAREHOUSENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

USEDORUNUSED
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   PORTS

PORTNAME
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRY
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFDOCKS
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFSHIPSATPORT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   DOCKS

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

OCCUPIEDORNOTOCCUPIED
INDEX =  FALSE  CLUSTERED =  FALSE

PORTNAME←DOCKNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   STOPS

SHIPID
```

```
INDEX =   TRUE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

PORTNAME
INDEX =  FALSE  CLUSTERED =  FALSE

ARRIVALDATE
INDEX =  FALSE  CLUSTERED =  FALSE

ARRIVALTIME
INDEX =  FALSE  CLUSTERED =  FALSE

DEPARTUREDATE
INDEX =  FALSE  CLUSTERED =  FALSE

DEPARTURETIME
INDEX =  FALSE  CLUSTERED =  FALSE

DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =   TRUE  CLUSTERED =  FALSE

PORTNAME←DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   LEGS

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SOURCESTOP
INDEX =  FALSE  CLUSTERED =  FALSE

DESTINATIONSTOP
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←SOURCESTOP
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←DESTINATIONSTOP
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   VOYAGES

SHIPID
INDEX =   TRUE  CLUSTERED =  FALSE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CHARTERER
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER
INDEX =   TRUE  CLUSTERED =  FALSE


RELATION   CARGOCLASSES

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

WUNIT
INDEX =  FALSE  CLUSTERED =  FALSE

VUNIT
```

```
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPCLASSCARGOCLASS

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

MAXVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

MAXWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   COUNTRIES

COUNTRYNAME
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRYABB
INDEX =   TRUE  CLUSTERED =   TRUE

POPULATION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPS

SHIPNAME
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPID
INDEX =   TRUE  CLUSTERED =  FALSE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

IRCS
INDEX =  FALSE  CLUSTERED =  FALSE

HULLNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

OWNER
INDEX =   TRUE  CLUSTERED =   TRUE

COUNTRYOFREGISTRY
INDEX =   TRUE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

DATEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

TIMEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

ATPORTORSEA
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPCLASSES

SHIPTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

WCAP
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
VCAP
INDEX =  FALSE  CLUSTERED =  FALSE

CREWSZ
INDEX =  FALSE  CLUSTERED =  FALSE

LIFEBOATCAP
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCAP
INDEX =  FALSE  CLUSTERED =  FALSE

CRUISESPD
INDEX =  FALSE  CLUSTERED =  FALSE

MAXSPD
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATMAX
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATCRUISING
INDEX =  FALSE  CLUSTERED =  FALSE

BEAM
INDEX =  FALSE  CLUSTERED =  FALSE

LENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

DEADWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPTYPES

SHIPTYPE
INDEX =   TRUE  CLUSTERED =   TRUE

DESCRIPTION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   FUELTYPES

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

PRICE
INDEX =  FALSE  CLUSTERED =  FALSE

UNIT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
OPTIMAL ACCESS CONFIGURATION FOR SITUATION 72
TOTALCOST =  8.861027836E+05  NOT CORRECT IF STEP1COST WAS USED

RELATION   TRACKS

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

DATE
INDEX =  FALSE  CLUSTERED =  FALSE

TIME
INDEX =  FALSE  CLUSTERED =  FALSE

COURSE
INDEX =  FALSE  CLUSTERED =  FALSE

SPEED
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

REPORTER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   CARGOESONBOARD

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   LOADEDUNLOADEDCARGOES

SHIPID
INDEX =  FALSE  CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

LORU
INDEX =  FALSE  CLUSTERED =  FALSE

QTYWGHT
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
QTYVOL
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   WAREHOUSES

PORTNAME
INDEX =   TRUE  CLUSTERED =   TRUE

WAREHOUSENUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

USEDORUNUSED
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE

QUANTITYVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   PORTS

PORTNAME
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRY
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFDOCKS
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

NUMBEROFSHIPSATPORT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   DOCKS

PORTNAME
INDEX =  FALSE  CLUSTERED =   TRUE

DOCKNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

MAXLENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

OCCUPIEDORNOTOCCUPIED
INDEX =  FALSE  CLUSTERED =  FALSE

PORTNAME←DOCKNUMBER
INDEX =   TRUE  CLUSTERED =   TRUE


RELATION   STOPS

SHIPID
```

```
INDEX =  FALSE   CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

STOPNUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

PORTNAME
INDEX =  FALSE   CLUSTERED =  FALSE

ARRIVALDATE
INDEX =  FALSE   CLUSTERED =  FALSE

ARRIVALTIME
INDEX =  FALSE   CLUSTERED =  FALSE

DEPARTUREDATE
INDEX =  FALSE   CLUSTERED =  FALSE

DEPARTURETIME
INDEX =  FALSE   CLUSTERED =  FALSE

DOCKNUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←STOPNUMBER
INDEX =    TRUE   CLUSTERED =   TRUE

PORTNAME←DOCKNUMBER
INDEX =  FALSE   CLUSTERED =  FALSE


RELATION   LEGS

SHIPID
INDEX =  FALSE   CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

LEGNUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

SOURCESTOP
INDEX =  FALSE   CLUSTERED =  FALSE

DESTINATIONSTOP
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←LEGNUMBER
INDEX =    TRUE   CLUSTERED =   TRUE

SHIPID←VOYAGENUMBER←SOURCESTOP
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER←DESTINATIONSTOP
INDEX =  FALSE   CLUSTERED =  FALSE


RELATION   VOYAGES

SHIPID
INDEX =  FALSE   CLUSTERED =   TRUE

VOYAGENUMBER
INDEX =  FALSE   CLUSTERED =  FALSE

CHARTERER
INDEX =  FALSE   CLUSTERED =  FALSE

SHIPID←VOYAGENUMBER
INDEX =    TRUE   CLUSTERED =   TRUE


RELATION   CARGOCLASSES

CARGOCLASS
INDEX =  FALSE   CLUSTERED =  FALSE

WUNIT
INDEX =  FALSE   CLUSTERED =  FALSE

VUNIT
```

```
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPCLASSCARGOCLASS

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

CARGOCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

MAXVOLUME
INDEX =  FALSE  CLUSTERED =  FALSE

MAXWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   COUNTRIES

COUNTRYNAME
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRYABB
INDEX =   TRUE  CLUSTERED =   TRUE

POPULATION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPS

SHIPNAME
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPID
INDEX =   TRUE  CLUSTERED =   TRUE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

IRCS
INDEX =  FALSE  CLUSTERED =  FALSE

HULLNUMBER
INDEX =  FALSE  CLUSTERED =  FALSE

OWNER
INDEX =  FALSE  CLUSTERED =  FALSE

COUNTRYOFREGISTRY
INDEX =  FALSE  CLUSTERED =  FALSE

LATITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

NORS
INDEX =  FALSE  CLUSTERED =  FALSE

LONGITUDE
INDEX =  FALSE  CLUSTERED =  FALSE

EORW
INDEX =  FALSE  CLUSTERED =  FALSE

DATEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

TIMEREPORTED
INDEX =  FALSE  CLUSTERED =  FALSE

ATPORTORSEA
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPCLASSES

SHIPTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

SHIPCLASS
INDEX =  FALSE  CLUSTERED =  FALSE

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

WCAP
INDEX =  FALSE  CLUSTERED =  FALSE
```

```
VCAP
INDEX =  FALSE  CLUSTERED =  FALSE

CREWSZ
INDEX =  FALSE  CLUSTERED =  FALSE

LIFEBOATCAP
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCAP
INDEX =  FALSE  CLUSTERED =  FALSE

CRUISESPD
INDEX =  FALSE  CLUSTERED =  FALSE

MAXSPD
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATMAX
INDEX =  FALSE  CLUSTERED =  FALSE

FUELCONSATCRUISING
INDEX =  FALSE  CLUSTERED =  FALSE

BEAM
INDEX =  FALSE  CLUSTERED =  FALSE

LENGTH
INDEX =  FALSE  CLUSTERED =  FALSE

MAXDRAFT
INDEX =  FALSE  CLUSTERED =  FALSE

DEADWEIGHT
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   SHIPTYPES

SHIPTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

DESCRIPTION
INDEX =  FALSE  CLUSTERED =  FALSE


RELATION   FUELTYPES

FUELTYPE
INDEX =  FALSE  CLUSTERED =  FALSE

PRICE
INDEX =  FALSE  CLUSTERED =  FALSE

UNIT
INDEX =  FALSE  CLUSTERED =  FALSE
```

Figure K-8: Situations 70, 71, 72, and their optimal solutions.

References

[AND 77] Anderson, H. D., Berra, P. B.
Minimum Cost Selection of Secondary Indexes for Formatted Files.
ACM Trans. Database Systems 2(1):68-90, March, 1977.

[AST 76] Astrahan, M. M. et al.
System R: Relational Approach to Database Management.
ACM Trans. Database Systems 1(2):97-137, June, 1976.

[AST 80] Astrahan, M. M. et al.
Performance of the System R Access Path Selection Mechanism.
Information Processing :487-491, 1980.
IFIP, North-Holland Publishing Co.

[BAT 80] Batory, D. S. and Gotlieb, C. C.
A Unifying Model of Physical Databases.
Technical Report CSRG-109, Computer Systems Research Group, University of Toronto, April, 1980.

[BAT 82] Batory, D. S. and Gotlieb, C. C.
A Unifying Model of Physical Databases.
ACM Trans. Database Systems 7(4):509-539, December, 1982.

[BAY 72] Bayer, R. and McCreight, E.
Organization and Maintenance of Large Ordered Indices.
Acta Informatica 1, 1972.

[BER 81] Berstein, P. A. et al.
Query Processing in a System for Distributed Databases (SDD-1).
ACM Trans. Database Systems 6(4):602-625, December, 1981.

[BLA 76] Blasgen, M. W. and Eswaren, K. P.
On the Evaluation of Queries in a Database System.
IBM Research Report RJ1945, IBM, San Jose, Calif., April, 1976.

[BRO 82] Brodie, M. and Schmidt, J.
Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group.
Sigmod Record 12(4):1-62, July, 1982.

[CAR 75] Cardenas, A. F.
Analysis and Performance of Inverted Database Structures.
Comm. ACM 18(5):253-263, May, 1975.

[CHA 76] Chamberlin, D. D., et al.
SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control.
IBM J. Res. and Devel. 20(6):560-575, November, 1976.

REFERENCES

[COD 70] Codd, E. F.
A Relational Model of Data for Large Shared Data Banks.
Comm. ACM 13(6):377-387, June, 1970.

[COD 71] CODASYL.
Data Base Task Group Report.
ACM, New York, 1971.

[COD-a 78] CODASYL Data Description Language Committee.
Journal of Development.
EDP Standards Committee, Secretariat of Canadian Government, Canada, 1978.

[COD-b 78] CODASYL Cobol Committee.
Journal of Development.
EDP Standard Committee, Secretariat of Canadian Government, Canada, 1978.

[COM 78] Comer, D.
The Difficulty of Optimum Index Selection.
ACM Trans. Database Systems 3(4):440-445, December, 1978.

[COM 79] Comer, D.
The Ubiquitous B-Tree.
ACM Trans. Database Systems 11(2):121-137, June, 1979.

[DE 78] De, P. et al.
Towards an Optimal Design of a Network Database from Relation Descriptions.
Operations Research 26(5):805-823, Sept.-Oct., 1978.

[DEC 78] *DECsystem-10/DECSYSTEM-20 Hardware Reference Manual- Central Processor*
Digital Equipment Corporation, 1978.

[DEM 80] Demolombe, R.
Estimation of the Number of Tuples Satisfying a Query Expressed in Predicate Calculus Language.
In *Proc. Intl. Conf. on Very Large Databases*, pages 55-63. Montreal, Canada, 1980.

[ELM 80] El-Masri, R. and Wiederhold G.
Properties of Relationships and Their Representation.
In *Natl. Computer Conf.*, pages 191-192. AFIPS, Vol. 49, May, 1980.

[FEL 66] Feldman, E., et al.
Warehouse Location Under Continuous Economies of Scale.
Management Science 12(9):670-684, July, 1966.

[FIN 82] Finkelstein, S. J. et al.
DBDSGN – A Physical Database Design Tool for System R.
Database Engineering 5(1):9-11, March, 1982.
IEEE Computer Society.

[GAM 77] Gambino, T. J. and Gerritsen, R.
A Database Design Decision Support System.
In *Proc. Intl. Conf. on Very Large Databases*, pages 534-544. Tokyo, Japan, IEEE, October, 1977.

[GER 76] Gerritsen, R. et al.
WAND User's Guide.
Decision Sciences Working Paper 76-01-03, Wharton School, Univ. of Pennsylvania, 1976.

[GER 77] Gerritsen, R. et al.
Cost Effective Databse Design: An Integrated Model.
Decision Sciences Working Paper 77-12-03, Wharton School, Univ. of Pennsylvania, 1977.

[GOO 79] Goodman, N. et al.
Query Processing in SDD-1: A System for Distributed Databases.
Technical Report, Computer Corporation of America, 1979.

[GOT 75] Gotlieb, L.
Computing Joins of Relations.
In *Proc. Intl. Conf. on Management of Data*, pages 55-63. San Jose, Calif., May, 1975.

[HAM 76] Hammer, M. and Chan, A.
Index Selection in a Self-Adaptive Database Management System.
In *Proc. Intl. Conf. on Management of Data*, pages 1-8. Washington, D.C., ACM SIGMOD, June, 1976.

[HEV 79] Hevner, A. R., and Yao, S. B.
Query Processing in Distributed Database Systems.
IEEE Trans. Software Eng. SE-5:177-187, May, 1979.

[HON 71] Honeywell Information Systems Inc.
Integrated Data Store
1971.

[HSI 70] Hsiao, D. and Harary, F.
A Formal System for Information Retrieval from Files.
Comm. ACM 13(4):67-73, February, 1970.
Also see Comm. ACM 13, 4, April 1070, p.266.

[IBM 70] IBM.
System 360 Scientific Subroutine Package
1970.

[KAT 80] Katz, R. H. and Wong, E.
An Access Path Model for Physical Database Design.
In *Proc. Intl. Conf. on Management of Data*, pages 22-29. Santa Monica, Calif., ACM SIGMOD, May, 1980.

REFERENCES

[KEL 81] Keller, A. M.
Updates to Relational Databases Through Views Involving Joins.
IBM Research Report RJ3282, IBM, San Jose, Calif., October, 1981.

[KIM 79] Kim, W.
Relational Database Systems.
ACM Computing Surveys 11(3):185-212, September, 1979.

[KIM 82] Kim, W.
On Optimizing an SQL-like Nested Query.
ACM Trans. Database Systems 7(3):443-469, September, 1982.

[KIN 74] King, W. F.
On the Selection of Indices for a File.
IBM Research Report RJ1341, IBM, San Jose, Calif., 1974.

[KNU-a 73] Knuth, D.
The Art of Computer Programming – Fundamental Algorithms.
Addeson-Wesley, 1973.

[KNU-b 73] Knuth, D.
The Art of Computer Programming – Sortng and Searching.
Addison-Wesley, 1973.

[KOO 82] Kooi, R. and Frankforth, D.
Query Optimization in INGRES.
Database Engineering 5(3):2-5, September, 1982.
IEEE Computer Society.

[KUE 63] Kuehn, A. A. and Hamburger, M. J.
A Heuristic Program for Locating Warehouses.
Management Science 10:643-657, July, 1963.

[LUK 83] Luk, W. S.
On Estimating Block Accesses in Database Organizations.
Comm. ACM 26, 1983.

[LUM 71] Lum, V. Y. and Ling, H.
An Optimization Problem of the Selection of Secondary Keys.
In *ACM Natl. Conf.*, pages 349-356. ACM, 1971.

[LUM 78] Lum, V. et al.
1978 New Orleans Data Base Design Workshop Report.
IBM Research Report RJ2554, IBM, San Jose, Calif., July, 1978.

[MIT 75] Mitoma, M. F. and Irani, K. B.
Automatic Data Base Schema Design and Optimization.
In *Proc. Intl. Conf. on Very Large Databases.* Framingham, Mass., September, 1975.

REFERENCES

[PEC 75] Pecherer, R. M.
Efficient Evaluation of Expression in a Relational Algebra.
In *ACM Pacific 75 Regional Conference*, pages 44-49. San Francisco, April, 1975.

[ROT 74] Rothnie, J. B. and Lozano, T.
Attribute Based File Organization in a Paged Memory Environment.
Comm. ACM 17(2):63-69, February, 1974.

[SCH 75] Schkolnick, M.
The Optimal Selection of Secondary Indices for Files.
Information Systems 1:141-146, March, 1975.

[SCH 79] Schkolnick, M. and Tiberio, P.
Considerations in Developing a Design Tool for a Relational DBMS.
In *Proc. Intl. Comp. Soft. & Appl. Conf.*, pages 228-235. Chicago, IEEE, November, 1979.

[SCH 81] Schkolnick, M. and Tiberio, P.
A Note on Estimating the Maintenance Cost in a Relational Database.
IBM Research Report RJ3327, IBM, San Jose, Calif., December, 1981.

[SEL 79] Selinger, P. G. et al.
Access Path Selection in a Relational Database Management System.
In *Proc. Intl. Conf. on Management of Data*, pages 23-34. Boston, Mass., May, 1979.

[SEN 69] Senko, M. E., et al.
File Design Handbook
IBM San Jose Research Laboratory, 1969.

[SEV 72] Severance, D. G.
Some Generalized Modeling Structures for Use in Design of File Organizations.
PhD thesis, University of Michigan, Ann Arbor, Mich., 1972.

[SEV 75] Severance, D. G.
A Parametric Model of Alternative File Structures.
Information Systems 1(2):51-55, 1975.

[SEV 77] Severance, D. G. and Carlis, J. V.
A Practical Approach to Selecting Record Access Paths.
ACM Computing Surveys 9(4):259-272, December, 1977.

[SIL 76] Siler, K. F.
A Stochastic Evaluation Model for Database Organizations in Data Retrival Systems.
Comm. ACM 19(2):84-95, February, 1976.

[SMI 75] Smith, J. and Chang, P.
Optimizing the Performance of a Relational Algebra Database Interface.
Comm. ACM 18(10):568-579, October, 1975.

REFERENCES

[STO 74] Stonebraker, M.
The Choice of Partial Inversions and Combined Indices.
Int. Journal of Computer Information Sciences 3(2):167-188, 1974.

[STO 76] Stonebraker, M. et al.
The Design and Implementation of INGRES.
ACM Trans. Database Systems 1(3):189-222, September, 1976.

[TEO 78] Teorey, T. J. and Oberlander, L. B.
Network Database Evaluation Using Analytical Modelling.
In *Natl. Computer Conf.*. AFIPS, Anaheim, Calif., Vol. 47, 1978.

[ULL 82] Ullman J.
Principles of Database Systems.
Computer Science Press, Potomac, Maryland, 1982.

[WAT 72] Waters, S. J.
File Design Fallacies.
The Computer Journal 15(1):1-4, 1972.

[WAT 75] Waters, S. J.
Estimating Magnetic Disc Seeks.
The Computer Journal 18(1):12-17, 1975.

[WAT 76] Waters, S. J.
Hit Ratios.
The Computer Journal 19(1):21-24, 1976.

[WHA-a 81] Whang, K., Wiederhold, G., and Sagalowicz, D.
Separability: An Approach to Physical Database Design.
In *Proc. Intl. Conf. on Very Large Databases*, pages 320-332. Cannes, France, IEEE, September, 1981.

[WHA-a 82] Whang, K., Wiederhold, G., Sagalowicz, D.
Estimating Block Accesses in Database Organizations – A Closed Noniterative Formula.
, to appear in the Communications of the ACM (accepted), 1982.

[WHA-b 81] Whang, K., Wiederhold, G., Sagalowicz, D.
Separability as an Approach to Physical Database Design.
Technical Report STAN-CS-81-898, Stanford University, October, 1981.
Also numbered as CSL TR-222.

[WHA-b 82] Whang, K., Wiederhold, G., and Sagalowicz, D.
Physical Design of Network Model Databases Using the Property of Separability.
In *Proc. Intl. Conf. on Very Large Databases*, pages 98-107. Mexico City, Mexico, September, 1982.

REFERENCES

[WIE 79] Wiederhold, G. and El-Masri, R.
The Structural Model for Database Design.
In *Proc. Intl. Conf. on Entity Relationship Approach*, pages 247-267. Los Angeles, Calif., December, 1979.

[WIE 83] Wiederhold, G.
Database Design.
McGraw-Hill Book Company, New York, 1983.
second edition.

[WON 76] Wong, E. and Youseffi, K.
Decomposition – A Strategy for Query Processing.
ACM Trans. Database Systems 1(3):223-241, September, 1976.

[YAO 78] Yao, S. B. and DeJong, D.
Evaluation of Database Access Paths.
In *Proc. Intl. Conf. on Management of Data*, pages 66-67. Austin, Texas, ACM SIGMOD, June, 1978.

[YAO 79] Yao, S. B.
Optimization of Query Evaluation Algorithm.
ACM Trans. Database Systems 4(2):133-155, June, 1979.

[YAO-a 77] Yao, S. B.
An Attribute Based Model for Database Access Cost Analysis.
ACM Trans. Database Systems 2(1):45-67, March, 1977.

[YAO-b 77] Yao, S. B.
Approximating Block Accesses in Database Organizations.
Comm. ACM 20(4):260-261, 1977.

[YUE 75] Yue, P. C. and Wong, C. K.
Storage Cost Considerations in Secondary Index Selection.
International Journal of Computer and Information Sciences 4(4):307-327, 1975.